Predator’s kill switch: undocumented anti-analysis techniques in iOS spyware

By Jamf Threat Labs

Archived: 2026-04-10 03:15:51 UTC

A deep dive into the error code taxonomy and detection mechanisms that prior research didn't cover.

January 14 2026 by

Jamf Threat Labs

By: Shen Yuan and Nir Avraham

Introduction

In December 2025, Google's Threat Intelligence Group (GTIG) published extensive research on Intellexa's Predator spyware, documenting its zero-day exploit chains and the PREYHUNTER stager component. Their research identified that the "watcher" module detects developer mode, jailbreak tools, security applications and network interception configurations.

However, while conducting independent reverse engineering of a Predator sample, Jamf Threat Labs discovered several undocumented mechanisms that reveal how sophisticated this spyware's anti-analysis capabilities truly are. This post presents original findings including:

- A complete error code taxonomy (301-311) that enables operators to diagnose exactly why an implant failed
- Implementation details for each detection method that go beyond high-level descriptions
- An undocumented crash reporter monitoring system for anti-forensics
- SpringBoard hooking to hide recording indicators from victims
- Kernel exploitation class names that reveal the internal architecture

These findings demonstrate that Predator's operators have granular visibility into failed deployments — a capability that has significant implications for researchers attempting to analyze these samples.

The CSWatcherSpawner architecture

The implant contains a C++ class named `CSWatcherSpawner::CSWatcherSpawner` that orchestrates all anti-analysis checks. The class implements a comprehensive set of detection methods with a sophisticated reporting mechanism.

What makes this architecture notable is not just the breadth of checks, but the reporting mechanism that provides operators with precise diagnostic information when deployment fails.

Figure 1: check_perform() entry point showing getCountNames() check and error code 311 dispatch for multiple instance detection

The error code taxonomy

The most significant undocumented finding is Predator's error code system. When any anti-analysis check triggers, the malware doesn't simply terminate — it reports a specific error code to its command-and-control infrastructure before cleaning up and exiting.

The `check_perform()` function reveals the complete taxonomy:

Code	Trigger condition	Detection method
311	Multiple instances running	`getCountNames() >= 2`
310	Debug console attached	`is_console_attached()`
309	Device in restricted region	`Locale == "US" || "IL"`
301	Developer mode OR jailbreak	`is_developer() || is_not_phone()`
304	Security/analysis tools running	`is_unsafe_running()`
307	HTTP proxy configured	`is_proxy_running()`
308	Custom root CA installed	`is_rootca_installed()`

Table 1: Error codes and their conditions

The missing error codes: 302, 303, 305, 306

A careful examination of the binary's string section reveals an interesting gap in the error code taxonomy. The error codes stored in the `__cstring` section appear sequentially, but with notable gaps:

Figure 2: Error code strings in __cstring section — note the sequential addresses but non-sequential codes (301, 304, 307, 308) revealing missing 302, 303, 305, 306

Notably, error codes 302, 303, 305, and 306 are completely absent from this sample. These gaps in the numbering scheme suggest several possibilities:

- **Reserved codes:** Placeholder codes for future functionality or checks not yet implemented
- **Variant-specific:** These codes may be used in other Predator variants or versions targeting different platforms
- **Deprecated checks:** Detection methods that were removed but whose error codes were preserved in the taxonomy
- **Shared taxonomy:** The error code system shared across multiple Intellexa products, with different products implementing different subsets

The non-sequential numbering (jumping from 301 to 304, and from 304 to 307) indicates this error taxonomy has evolved over time or is designed to accommodate checks that may be conditionally compiled based on the target configuration.

Why this matters

This error code system transforms failed deployments from black boxes into diagnostic events. When an operator deploys Predator against a target and receives error code 304, they know the target is running security tools — not that the exploit failed, not that the device is incompatible, but specifically that active analysis is occurring.

This has direct implications for targeted individuals: if security analysis tools like Frida are running, Predator will abort deployment and report error code 304 to operators, who can then troubleshoot why their deployment failed.

Detection implementation details

While Google's research mentioned that Predator detects "custom HTTP proxies" and "custom root CAs," the actual implementation details were not published. Here's what the binary reveals:

Multiple instance detection (error 311)

The first check in `check_perform()` calls `getCountNames()` to detect if multiple Predator instances are running. This prevents researchers from running multiple analysis instances simultaneously:

```
__int64 __fastcall Utils::getCountNames(Utils *this)
{
  pid_t *v1; // x19
  __int64 v2; // x20
  size_t v4; // x23
  pid_t *v5; // x24
  int v6; // w22
  char buffer[4096]; // [xsp+8h] [xbp-1058h] BYREF
  int v8[2]; // [xsp+1008h] [xbp-58h] BYREF
  int v9; // [xsp+1010h] [xbp-50h]
  size_t v10; // [xsp+1018h] [xbp-48h] BYREF

  __chkstk_darwin(this);
  v10 = 0;
  *(_QWORD *)v8 = 0xE00000001LL;
  v9 = 0;
  if ( sysctl(v8, 3u, 0, &v10, 0, 0) < 0 )
    return 0xFFFFFFFFLL;
  v1 = (pid_t *)malloc(v10);
  if ( sysctl(v8, 3u, v1, &v10, 0, 0) < 0 )
    return 0xFFFFFFFFLL;
  if ( v10 >= 0x288 )
  {
    v2 = 0;
    v4 = v10 / 0x288;
    v5 = v1 + 10;
    do
    {
      v6 = *v5;
      if ( (getpgid(*v5) & 0x80000000) == 0 && proc_pidpath(v6, buffer, 0x1000u) )
      {
        if ( strstr(buffer, "/private/var/tmp/") )
          v2 = (unsigned int)(v2 + 1);
        else
          v2 = (unsigned int)v2;
      }
      v5 += 162;
      --v4;
    }
    while ( v4 );
  }
  else
  {
    v2 = 0;
  }
  free(v1);
  return v2;
}
```

Figure 3: getCountNames() pseudocode — enumerates all processes via sysctl, uses proc_pidpath to get executable paths, counts processes running from /private/var/tmp/

The function iterates through all running processes using `sysctl`, retrieves each process's executable path via `proc_pidpath()`, and counts how many are running from `/private/var/tmp/` (Predator's staging directory). If the count is two or more, error code 311 is reported.

Developer mode detection: targeting researchers

Building on Google's finding that Predator detects Developer Mode, we reverse engineered how this check works.

The `is_developer()` function specifically targets iOS Developer Mode using the `sysctlbyname` API:

Figure 4: Developer Mode detection implementation showing sysctlbyname("security.mac.amfi.developer_mode_status") call

This is significant because Developer Mode was introduced in iOS 16 specifically for security researchers and developers. By detecting this, Predator effectively says: "If you've enabled developer features, you're probably not a normal target."

Jailbreak detection: is_not_phone()

The `is_not_phone()` function checks for the presence of jailbreak-related files and directories:

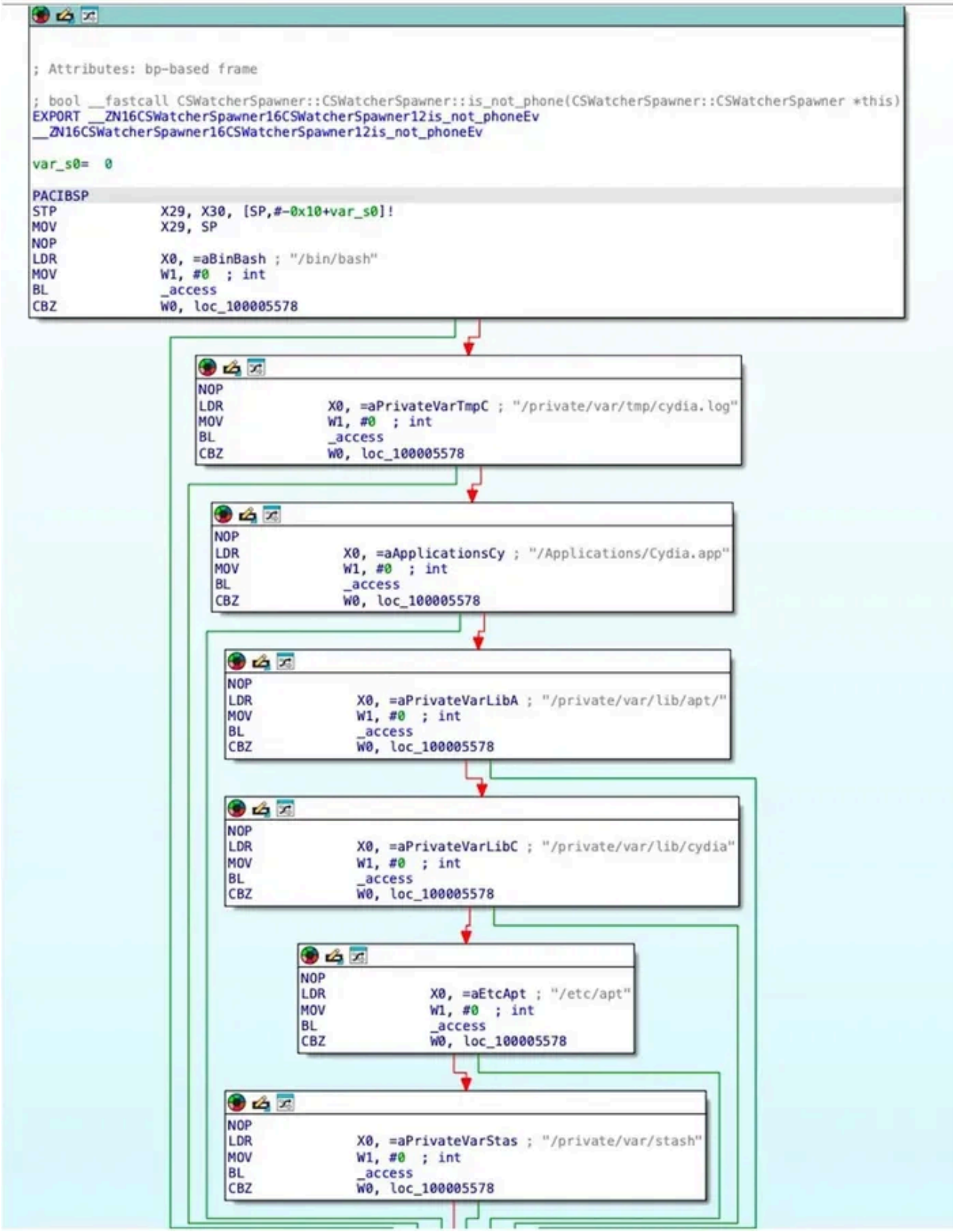

Figure 5: is_not_phone() graph view — cascading file existence checks for jailbreak indicators

The function checks for the following paths using the `access()` syscall:

- `/bin/bash` — Shell binary (not present on stock iOS)

- `/private/var/tmp/cydia.log` — Cydia package manager log
- `/Applications/Cydia.app` — Cydia application
- `/private/var/lib/apt/` — APT package manager directory
- `/private/var/lib/cydia` — Cydia data directory
- `/etc/apt` — APT configuration
- `/private/var/stash` — Jailbreak stash directory

Additionally, it checks if `/bin/` contains more than two entries (stock iOS has minimal binaries in /bin).

Geographic restrictions: avoiding US and Israeli jurisdiction

Google's research identified that Predator checks for US and Israeli locales. Our analysis documented the specific implementation of the geographic check (error code 309), which deserves further attention. Predator refuses to execute on devices with US or Israeli locale settings:

```
__text:00000001000059B0 loc_100005980                   ; CODE XREF: CSWatcherSpawner::CSWatcherSpawner::check_perform(void)+7C↑j
__text:00000001000059B0                 NOP
__text:00000001000059B4                 LDR             X0, =_OBJC_CLASS_$_NSLocale
__text:00000001000059B8                 BL              _objc_msgSend$autoupdatingCurrentLocale ; +[NSLocale autoupdatingCurrentLocale] ...
__text:00000001000059BC                 BL              _objc_msgSend$countryCode ; -[NSLocale countryCode] ...
__text:00000001000059C0                 BL              _objc_msgSend$UTF8String ; -[NSString UTF8String] ...
__text:00000001000059C4                 ADR             X8, aUs ; "US"
__text:00000001000059C8                 NOP
__text:00000001000059CC                 CMP             X0, X8
__text:00000001000059D0                 ADR             X8, aIl ; "IL"
__text:00000001000059D4                 NOP
__text:00000001000059D8                 CCMP            X0, X8, #4, NE
__text:00000001000059DC                 B.NE            loc_100005A10
__text:00000001000059E0                 LDR             X20, [X19,#0x110]
__text:00000001000059E4                 MOV             X0, X20 ; __s
__text:00000001000059E8                 BL              _strlen
__text:00000001000059EC                 ADD             X0, X0, #1 ; __count
__text:00000001000059F0                 MOV             W1, #1  ; __size
__text:00000001000059F4                 BL              _calloc
__text:00000001000059F8                 MOV             X21, X0
__text:00000001000059FC                 MOV             X1, X20 ; __src
__text:0000000100005A00                 BL              _strcpy
__text:0000000100005A04                 ADR             X2, a309 ; "309"
__text:0000000100005A08                 NOP
__text:0000000100005A0C                 B               loc_100005A78
```

Figure 6: Geographic restriction check using NSLocale — comparing country code against "US" and "IL", triggering error 309

Enlarge image

While we cannot confirm, this is likely a deliberate decision to avoid legal exposure in jurisdictions with strong cybercrime laws and active law enforcement cooperation with Intellexa's European base.

Console detection: timing-based analysis

The `is_console_attached()` function uses a clever timing-based approach:

```
__text:0000000100005978 loc_100005978                   ; CODE XREF: CSWatcherSpawner::CSWatcherSpawner::check_perform(void)+44↑j
__text:0000000100005978                 BL              __ZN16CSWatcherSpawner16CSWatcherSpawner19is_console_attachedEv ; CSWatcherSpawner::CSWatcherSpawner::is_console_attached(void)
__text:000000010000597C                 CBZ             W0, loc_1000059B0
__text:0000000100005980                 LDR             X20, [X19,#0x110]
__text:0000000100005984                 MOV             X0, X20 ; __s
__text:0000000100005988                 BL              _strlen
__text:000000010000598C                 ADD             X0, X0, #1 ; __count
__text:0000000100005990                 MOV             W1, #1  ; __size
__text:0000000100005994                 BL              _calloc
__text:0000000100005998                 MOV             X21, X0
__text:000000010000599C                 MOV             X1, X20 ; __src
__text:00000001000059A0                 BL              _strcpy
__text:00000001000059A4                 ADR             X2, a310 ; "310"
__text:00000001000059A8                 NOP
__text:00000001000059AC                 B               loc_100005A78
```

Figure 7: Debug console detection showing is_console_attached() call and error code 310 dispatch

Enlarge image

This detection compares the modification time of the diagnosticd filter file against the system boot time. If someone enabled console logging after the device booted (as a researcher would do), the file's mtime will be newer than the boot time.

Security tool detection: the complete list

Google's research mentioned "Bash, tcpdump, frida, sshd, or checkrain" but the actual list in the binary is longer. The `is_unsafe_running()` function reveals the complete process detection list:

Figure 8: is_unsafe_running() assembly — complete process detection list including tcpdump, frida-server, netstat, sshd, checkra1nd, loader, McAfee, AvastMobileSecurity, NortonMobileSecurity

Enlarge image

The complete list of detected processes also includes:

- `tcpdump` — network packet capture
- `frida-server` — dynamic instrumentation framework
- `netstat` — network statistics
- `sshd` — SSH daemon
- `checkra1nd` — checkra1n jailbreak
- `loader` — generic loader process
- `McAfee` — McAfee mobile security
- `AvastMobileSecurity` — Avast mobile security
- `NortonMobileSecurity` — Norton mobile security

The inclusion of netstat is noteworthy — it suggests Predator is concerned about targets who might be monitoring their own network connections, not just researchers with specialized tools. A privacy-conscious user simply checking their network connections would trigger this detection.

Self-cleanup after error reporting and shutdown

After reporting an error code to C2, Predator performs self-cleanup by removing its staging directory:

```
__text:0000000100005AB4
__text:0000000100005AB4 loc_100005AB4                   ; CODE XREF: CSWatcherSpawner::CSWatcherSpawner::check_perform(void)+19C↑j
__text:0000000100005AB4                 ADR             X0, _WatcherPathSecond ; "/private/var/tmp/"
__text:0000000100005AB8                 NOP
__text:0000000100005ABC                 BL              _remove
__text:0000000100005AC0                 MOV             W0, #0  ; exc_buf
__text:0000000100005AC4                 LDP             X29, X30, [SP,#0x30+var_s0]
__text:0000000100005AC8                 LDP             X20, X19, [SP,#0x30+var_10]
__text:0000000100005ACC                 LDP             X22, X21, [SP,#0x30+var_20]
__text:0000000100005AD0                 ADD             SP, SP, #0x40 ; '@'
__text:0000000100005AD4                 RETAB
```

Figure 9: Self-cleanup code calling _remove on /private/var/tmp/ staging directory after error reporting

This cleanup happens after the C2 callback, ensuring operators receive the diagnostic information even if the malware is immediately removed.

Another cleanup mechanism is tied to the device shutdown lifecycle, the implant registers a Darwin notification observer for “com.apple.springboard.deviceWillShutDown”.

```
1  void __fastcall CSWatcherSpawner::CSWatcherSpawner::listenForShutdown(CSWatcherSpawner::CSWatcherSpawner *this)
2  {
3    __CFNotificationCenter *DarwinNotifyCenter; // x0
4    __int64 vars8; // [xsp+18h] [xbp+8h]
5
6    DarwinNotifyCenter = CFNotificationCenterGetDarwinNotifyCenter();
7    if ( ((vars8 ^ (2 * vars8)) & 0x4000000000000000LL) != 0 )
8      __break(0xC471u);
9    CFNotificationCenterAddObserver(
10     DarwinNotifyCenter,
11     this,
12     (CFNotificationCallback)CSWatcherSpawner::CSWatcherSpawner::onShutdown,// callback
13     CFSTR("com.apple.springboard.deviceWillShutDown"),
14     0,
15     CFNotificationSuspensionBehaviorDeliverImmediately);
16 }
```

Figure 10: Code to register listener for shutdown signal

When the shutdown notification is received, it transits to a cleanup routine removes specific on-disk evidences if they exist.

```
void __fastcall CSWatcherSpawner::CSWatcherSpawner::stop(CFRunLoopTimerRef *this)
{
  std::error_code *v2; // x1
  std::error_code *v3; // x1
  __CFRunLoop *Main; // x0
  __CFRunLoop *v5; // x0
  __int64 vars8; // [xsp+18h] [xbp+8h]

  if ( access(BitPath, 0) )
  {
    if ( access(HelperPath, 0) )
      goto LABEL_3;
  }
  else
  {
    remove((const std::__fs::filesystem::path *)BitPath, v2);
    if ( access(HelperPath, 0) )
      goto LABEL_3;
  }
  remove((const std::__fs::filesystem::path *)HelperPath, v3);
LABEL_3:
  Main = CFRunLoopGetMain();
  CFRunLoopRemoveTimer(Main, this[49], kCFRunLoopDefaultMode);
  v5 = CFRunLoopGetMain();
  if ( ((vars8 ^ (2 * vars8)) & 0x4000000000000000LL) != 0 )
    __break(0xC471u);
  CFRunLoopStop(v5);
}
```

Figure 11: Self-cleanup code on shutdown process

Undocumented anti-forensics: crash reporter monitoring

One capability not mentioned in any public research is the `monitoringCrashReporter()` function:

```
// Anti-forensics: Monitors /private/var/mobile/Library/Logs/CrashReporter/ using kqueue for file system events.
__int64 __fastcall CSWatcherSpawner::CSWatcherSpawner::monitoringCrashReporter(
        CSWatcherSpawner::CSWatcherSpawner *this)
{
  int v1; // w0
  int v2; // w19
  void *v3; // x22
  kevent v5; // [xsp+0h] [xbp-70h] BYREF
  kevent changelist; // [xsp+20h] [xbp-50h] BYREF

  sleep(1u);
  v1 = kqueue();
  if ( (v1 & 0x80000000) == 0 )
  {
    v2 = v1;
    changelist.ident = open("/private/var/mobile/Library/Logs/CrashReporter/", 0);
    *(_QWORD *)&changelist.filter = 0xF0015FFFCLL;
    changelist.data = 0;
    changelist.udata = 0;
    while ( 1 )
    {
      while ( kevent(v2, &changelist, 1, &v5, 1, 0) < 0 )
        ;
      v3 = objc_autoreleasePoolPush();
      -[NSArray enumerateObjectsUsingBlock:](
        -[NSFileManager contentsOfDirectoryAtPath:error:](
          +[NSFileManager defaultManager](&OBJC_CLASS___NSFileManager, "defaultManager"),
          "contentsOfDirectoryAtPath:error:",
          CFSTR("/private/var/mobile/Library/Logs/CrashReporter/"),
          0),
        "enumerateObjectsUsingBlock:",
        &stru_100044A68);
      objc_autoreleasePoolPop(v3);
    }
  }
  return 0xFFFFFFFFLL;
}
```

Figure 12: monitoringCrashReporter() — kqueue-based monitoring of /private/var/mobile/Library/Logs/CrashReporter/

This function uses kqueue to monitor the CrashReporter directory for any new files. When a crash occurs that might expose Predator's presence, the handler processes or removes the crash log before it can be synced or examined.

Memory forensics suppression: killing mmaintenanced

The crash handler specifically targets memory forensics evidence:

```
// Block callback for crash reporter monitoring. Checks if crash report filename contains "SystemMemory" and kills mmaintenanced (signal 9) to suppress memory forensics evidence.
void __cdecl crash_reporter_block_handler(id a1, id a2, unsigned __int64 a3, bool *a4)
{
  const char *v4; // x1
  pid_t Pid; // w0
  __int64 _8; // [xsp+8h] [xbp+8h]

  if ( (unsigned int)objc_msgSend(a2, "containsString:", CFSTR("SystemMemory"), a4) )
  {
    Pid = Utils::getPid((Utils *)"mmaintenanced", v4);
    if ( Pid )
    {
      if ( ((_8 ^ (2 * _8)) & 0x4000000000000000LL) != 0 )
        __break(0xC471u);
      kill(Pid, 9);
    }
  }
}
```

Figure 13: crash_reporter_block_handler — kills mmaintenanced when SystemMemory crash reports are detected

Enlarge image

When a crash report filename contains "SystemMemory," the handler immediately locates and kills the `mmaintenanced` process (memory maintenance daemon) using SIGKILL (signal 9). This prevents investigators from capturing memory dumps.

This is significant for forensics: crash logs are a valuable artifact for detecting exploitation attempts, and Predator actively suppresses them.

Recording indicator hiding

The `TestHooker()` function reveals how Predator hides the iOS recording indicator from victims:

Figure 14: UMHooker using Mach exception handling for cross-process hooking into SpringBoard

Enlarge image

This code locates the SpringBoard process, uses kernel exploitation primitives to inject into SpringBoard, and hooks SBRecordingIndicatorManager methods to suppress the recording indicator. When Predator activates the microphone or camera, victims won't see the orange/green indicator dot.

Kernel exploitation class names

The code reveals several internal class names that indicate the exploitation architecture:

Table 2: Class names and their purposes

The `NSTaskROP::WithoutDeveloperMode` template is particularly interesting — it suggests Intellexa has developed ROP techniques that work even when developer mode is disabled, which is the default state for most users.

Stubbed functionality: is_corellium()

An interesting artifact is the `is_corellium()` function at address `0x100005bb8`:

Corellium is a cloud-based iOS device virtualization platform used by security researchers. The presence of this stubbed function suggests that Intellexa is aware of Corellium as an analysis platform; detection may have been implemented and later disabled, or detection is planned for future versions.

This function is not called from `check_perform()` in this sample, but its presence indicates awareness of the research community's tools.

Indicators of compromise

File access patterns

- `/private/var/preferences/SystemConfiguration/preferences.plist` (proxy check)
- `/private/var/preferences/Logging/com.apple.diagnosticd.filter.plist` (console check)
- `/private/var/protected/trustd/private/TrustStore.sqlite3` (root CA check)
- `/private/var/mobile/Library/Logs/CrashReporter/` (crash monitoring)

Jailbreak detection paths

Process detection targets

SQL queries (TrustStore)

Conclusion

This analysis reveals that Predator's anti-analysis capabilities are more sophisticated than previously documented. The error code taxonomy demonstrates that Intellexa operators have granular visibility into why deployments fail, enabling them to adapt their approaches for specific targets.

For researchers, these findings highlight the importance of:

- **Air-gapped analysis environments:** The C2 callback means any network-connected analysis will alert operators.
- **Crash log preservation:** Enable crash log collection before analysis begins.
- **Process name awareness:** Even running netstat triggers detection.
- **Boot-time considerations:** Console logging configured before boot may evade timing-based detection.

For the broader security community, this analysis demonstrates that commercial spyware vendors invest significant engineering effort into detecting researchers — not just evading security products. The presence of the `is_corellium()` stub shows they're watching our tools as closely as we're watching theirs.

Appendix: function reference

Function	Address	Description
`check_perform()`	0x100005900	Main orchestration, error code dispatch
`is_developer()`	0x1000055a8	sysctlbyname developer_mode_status
`is_not_phone()`	0x1000054b0	Jailbreak path detection
`is_unsafe_running()`	0x100005c10	Process name detection (incl. netstat)
`is_proxy_running()`	0x100005d60	SystemConfiguration.plist parsing
`is_rootca_installed()`	0x100005e3c	TrustStore.sqlite3 queries
`is_console_attached()`	0x10000579c	diagnosticd mtime comparison
`is_restricted_country()`	0x100005758	NSLocale country code check
`getCountNames()`	0x10000c85c	/private/var/tmp/ process counting
`is_corellium()`	0x100005bb8	Stubbed Corellium detection
`ReportAbort()`	0x100005620	C2 error reporting
`monitoringCrashReporter()`	0x1000067d4	kqueue crash log monitoring
`TestHooker()`	0x1000068fc	SpringBoard indicator hooking

Table 3: List of functions and associated address and descriptions

Dive into more Jamf Threat Labs research on our blog.

Tags:
