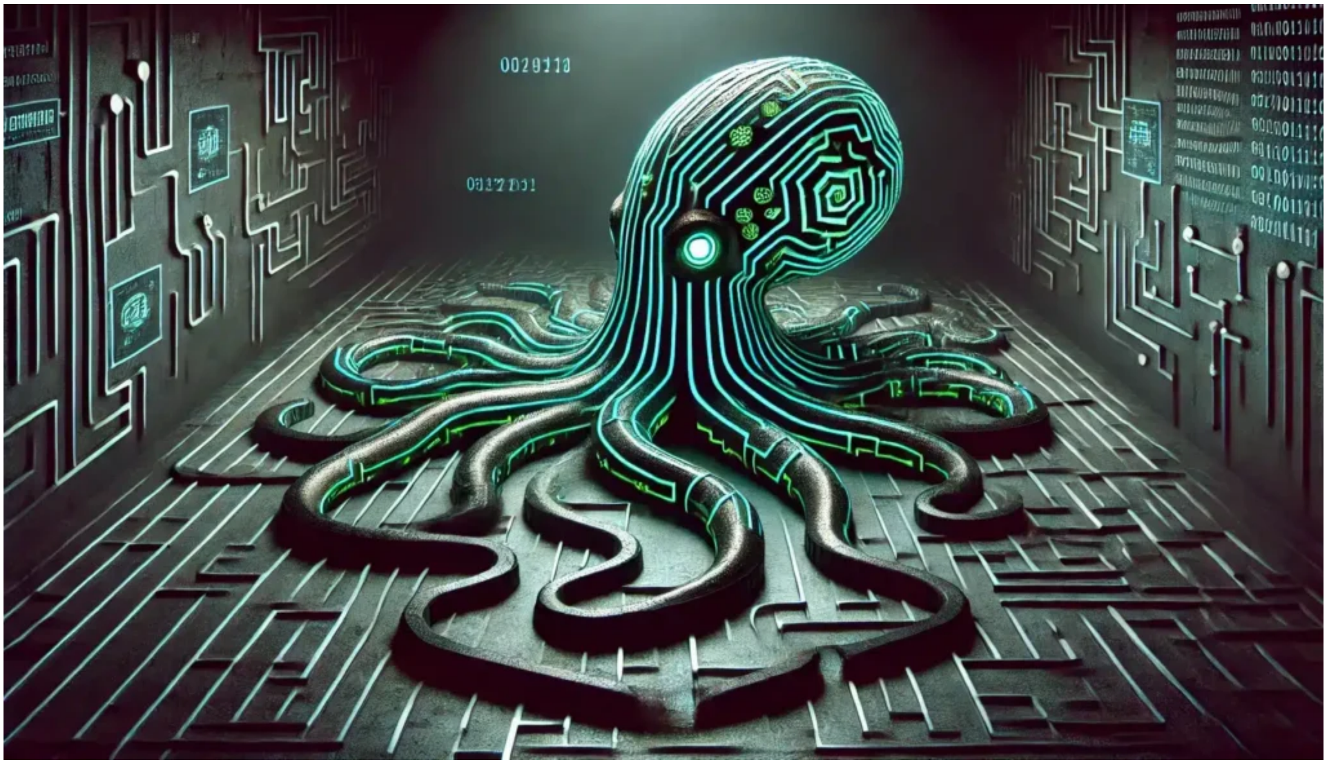
Babble Babble Babble Babble Babble Babble BabbleLoader

intezer.com/blog/research/babble-babble-babble-babble-babble-babble-babbleloader/

November 17, 2024



Loaders, an Ever Evolving Market

The pace of innovation and development in the malware detection market is relentless, the same goes for the development of malware itself. Constantly charging and adapting to create ever more evasive and capable payloads.

One such sector of this market is the loader (also called crypter or packer) market. In today's threat landscape, loaders have become a critical tool in cybercrime operations, serving as the backbone for delivering a range of malicious payloads. Loaders are often the first stage in an attack chain, designed to stealthily execute or inject malware, such as info-stealers or ransomware, into a target system. Their prevalence reflects an evolution in tactics, allowing threat actors to evade traditional antivirus defenses through techniques like in-memory execution and anti-analysis features. Widely available for purchase or lease on underground markets, loaders are now a commodity in malware distribution, making sophisticated attack methods accessible to a broader range of actors and adaptable across diverse campaigns and targets.

In this blog, we will introduce “BabbleLoader”, an extremely evasive loader, packed with defensive mechanisms, that is designed to bypass antivirus and sandbox environments to deliver stealers into memory.

BabbleLoader’s Techniques to Evade Traditional and AI Systems

BabbleLoader stands out for its array of sophisticated evasion techniques that challenge both traditional and AI-based detection systems. Key features include junk code insertion and metamorphic transformations, which alter the loader’s structure and flow, effectively evading signature-based, Artificial Intelligence, and behavioral detections. Through dynamic API resolution, the loader sidesteps common API monitoring by resolving necessary functions only at runtime, preventing static analysis from identifying telltale Windows APIs. Also bypassing sandbox injected DLLs that hook API calls. Shellcode loading and decryption further obfuscate the payload by embedding and decrypting malicious code in memory, bypassing file-based scanning. Additionally, anti-sandboxing and anti-analysis measures detect virtual environments, impeding sandbox analysis and automated AI defenses. Together, these techniques make this loader a versatile tool, capable of subverting both static and dynamic security layers.

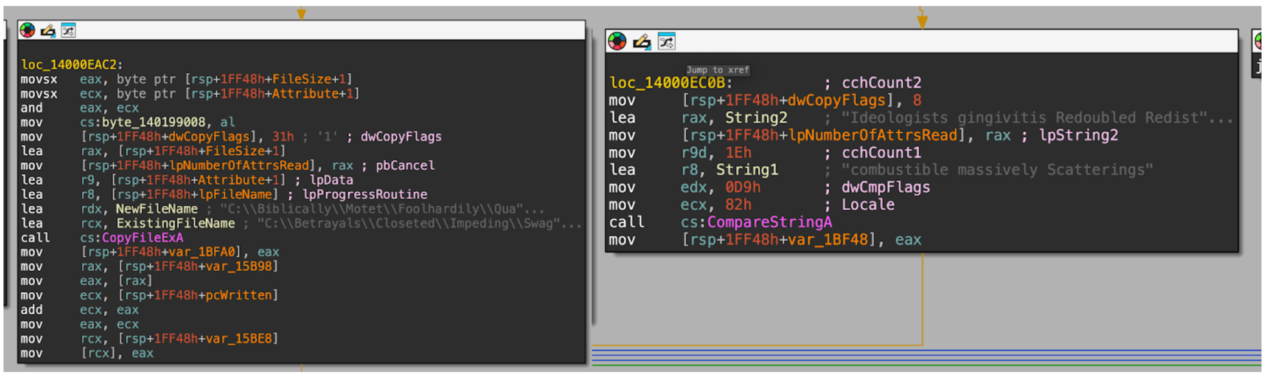
When investigating this loader, we have seen it used across multiple campaigns, targeting both English and Russian speaking individuals. Lure themes suggest it is targeting a vast range of users, from users looking to download generic cracked software, such as video editing, gaming, VPN, browsers, and utilities. We have also noticed campaigns that target with a particular focus on business professionals in finance and administration, masquerading as accounting software, and forms for filling out eligibility checks often used by HR or payroll professionals.

Technical Analysis

The sample used in this analysis is:
a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87

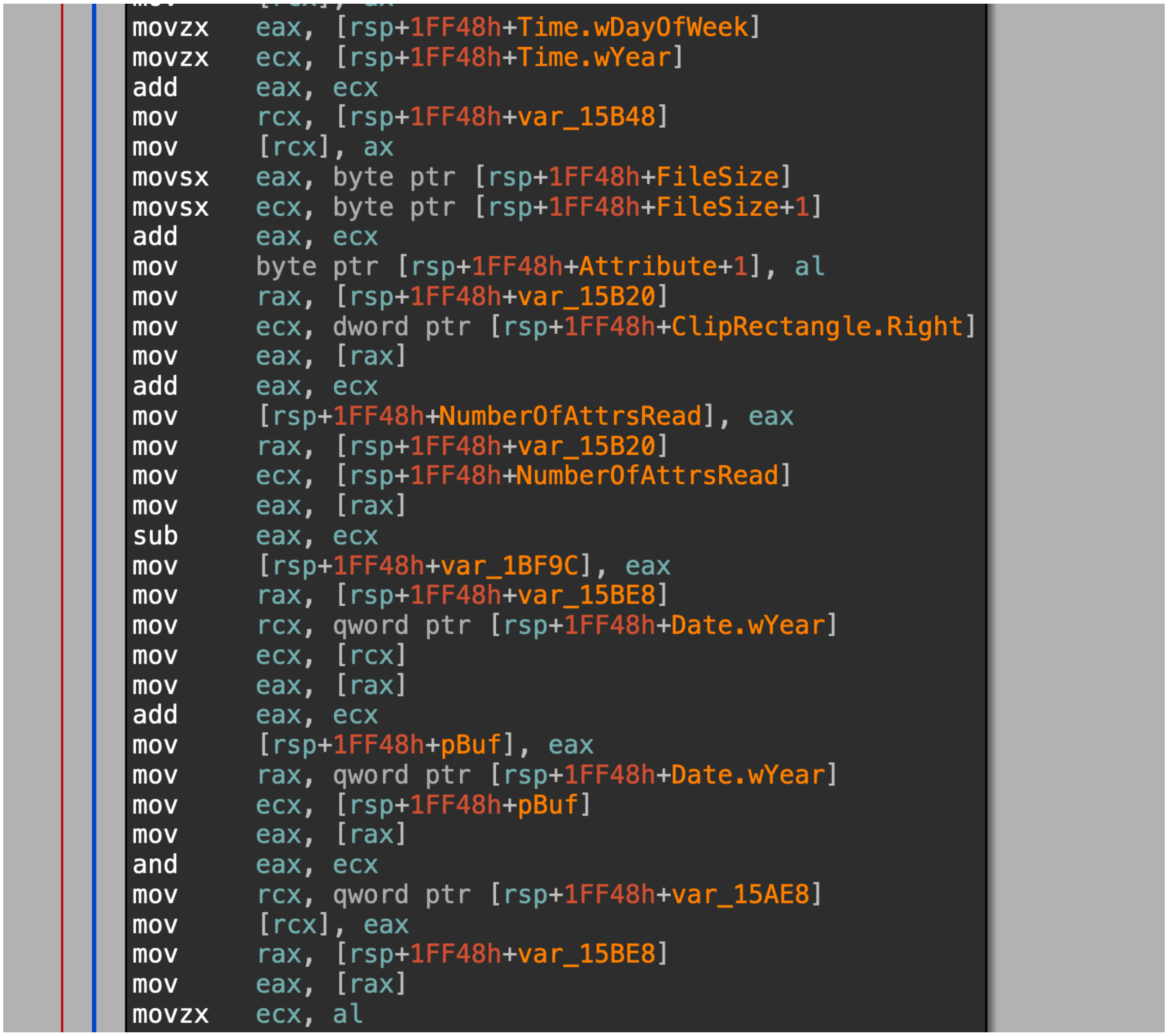
Junk Code/Metamorphism

BabbleLoader makes diabolical use of junk code. This is done in an effort to hamper analysis by confusing the analyst. This is achieved through multiple means. There are many paths of code that are never actually accessed, but use random imports with randomly generated hardcoded strings.



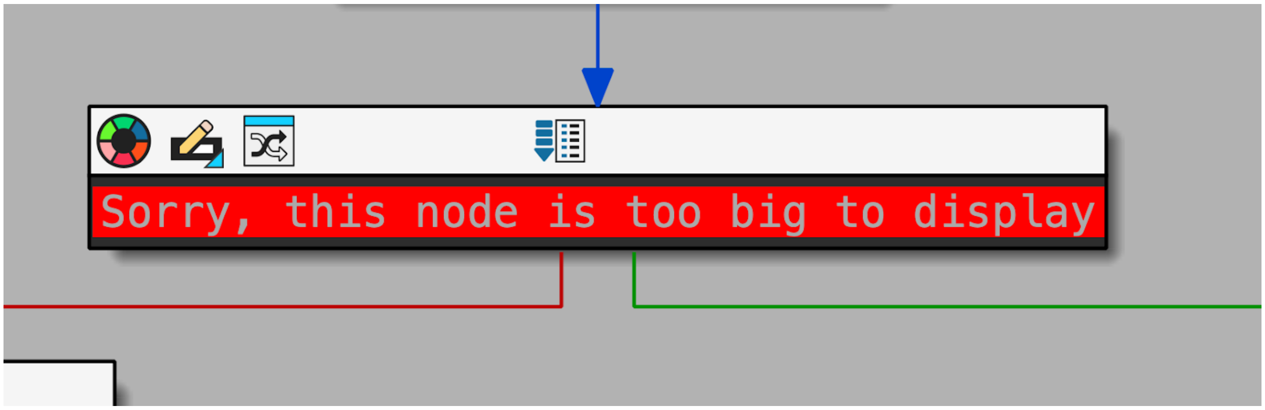
Junk Code making rubbish calls

The loader also makes excessive use of random instructions, adding values to local variables and moving data around registers for no particular functionality.



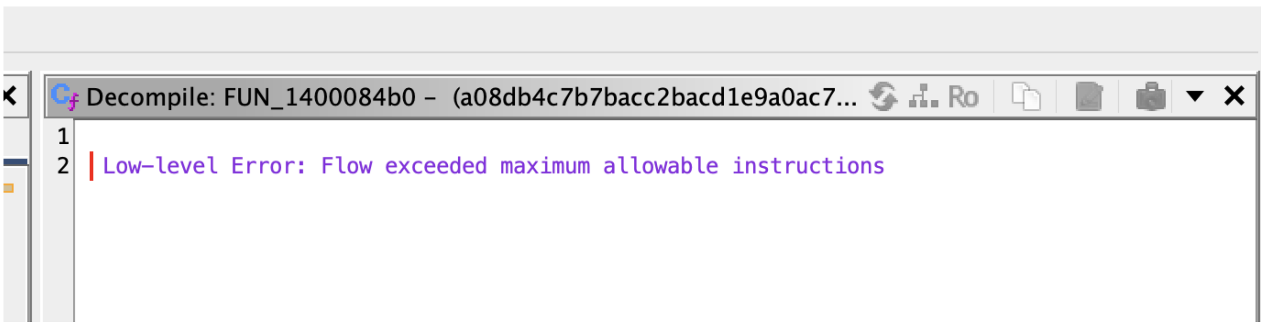
Junk Code

The amount of junk code added into the sample greatly increases the amount of code to the point where it starts to crash disassembly or decompilation tools through its sheer mass alone. In the case of IDA needs to collapse nodes due to them being so large.



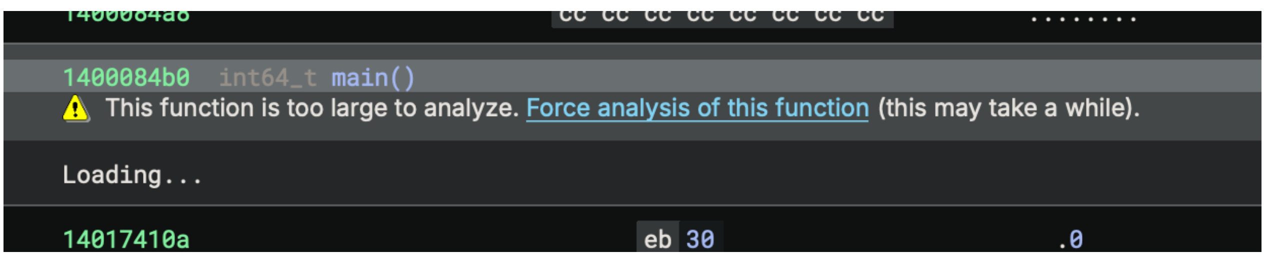
Collapsed Node in IDA

In Ghidra the function graph view will freeze and there are too many instructions for the decompiler to show.



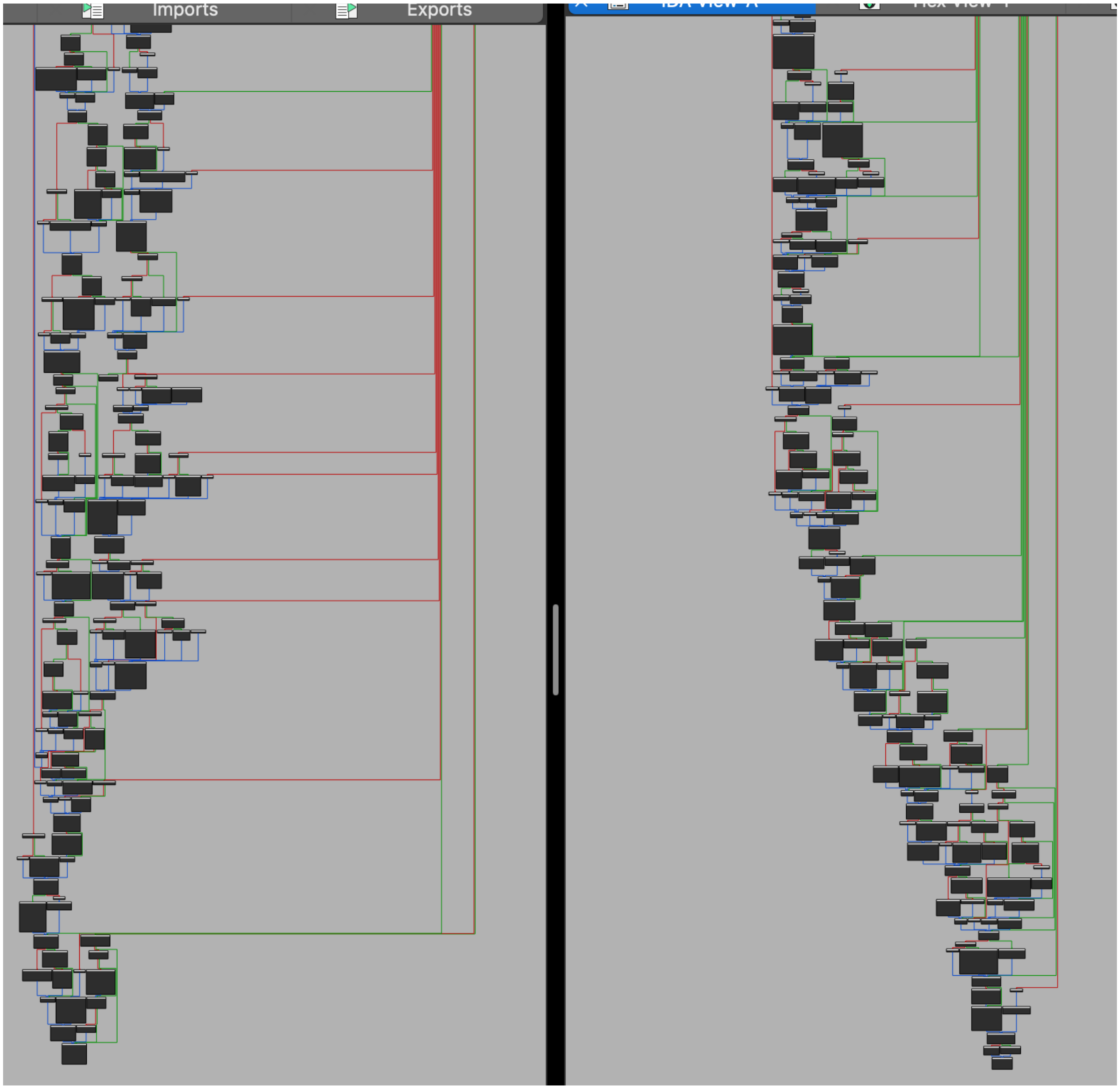
Decompilation output in Ghidra

We have even checked in Binary Ninja to see the effects of the junk code. The user is required to manually force analysis of the function due to the size.



Binary Ninja showing large function

Each of these techniques also serve the purpose of making the loader metamorphic. Each build of the loader will have unique strings, unique metadata, unique code, unique hashes, unique encryption, and a unique control flow. Each sample is structurally unique with only a few snippets of shared code. Below is a very small snippet of the main method of two different samples, showing very different control flow.



Comparison of structure of two BabbleLoader samples

Even the metadata of the file is randomized for each sample.

Property	Value
Comments	Ferry forbidden aniline tangle discoloured milkman
CompanyName	Outsourcing
FileDescription	Tormented cudgel sheer households drownings festivals
FileVersion	4.29.221.0
InternalName	Uprated disclaimer
LegalCopyright	Copyright © Saddle misunderstands respectable
LegalTrademarks	Babbling landmarks loveless metronomic

Junk Metadata

What This Means for AI-Based Analysis Techniques

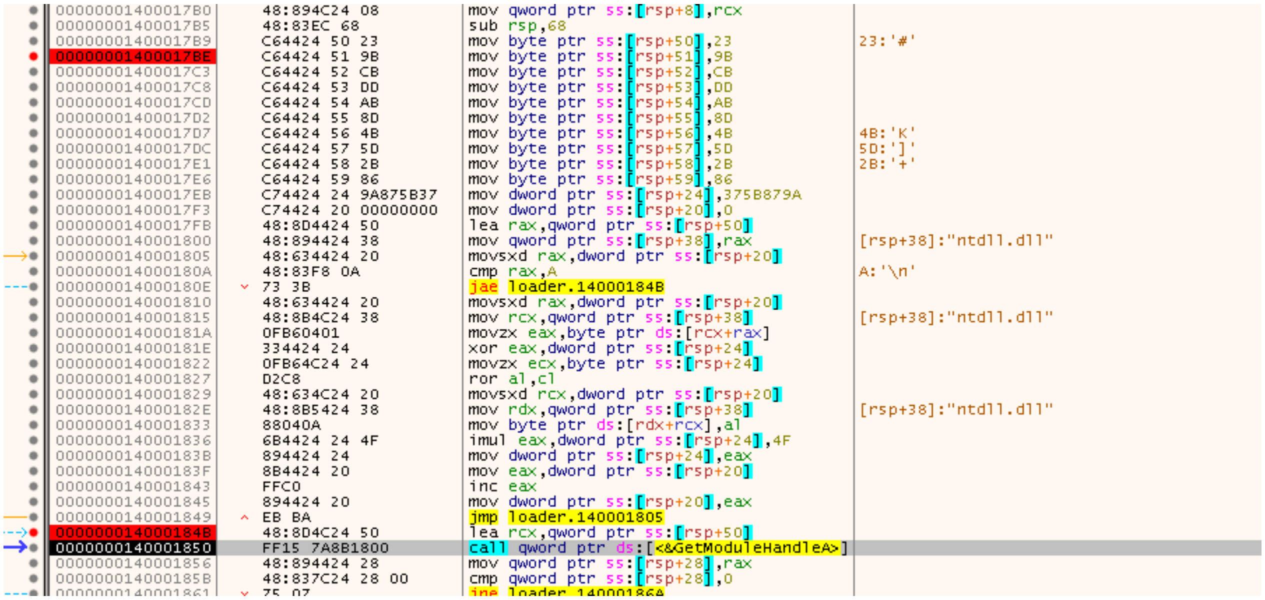
These techniques also have large implications for AI based analysis techniques. This constant variation in code structure forces AI models to continuously re-learn what to look for—a process that often leads to missed detections or false positives. By filling the code with junk instructions, the loader can trick AI into interpreting irrelevant actions as meaningful ones, leading it to predict that the malware will perform operations that it never actually executes. Junk code also generates a large volume of "noise" in the program flow, overwhelming the AI's pattern-recognition capabilities and forcing it to sift through thousands of extraneous actions that mask the true behavior of the malware.

Additionally, the inclusion of countless junk variables adds another layer of complexity. AI models analyzing variable behavior to understand data flow must now track thousands of decoy variables, each potentially obfuscated or dynamically transformed to further confuse the analysis. This variable noise, combined with the ever-shifting structure from metamorphism, makes it extremely difficult for AI to reliably determine which variables are integral to the malware's function and which are simply junk.

The sheer volume of junk code and variables also makes analyzing this loader exceptionally costly. The sheer number of tokens AI must process to parse and interpret the junk alone leads to high computational and financial costs, effectively weaponizing the malware's complexity against AI-driven defenses. This combination of overwhelming data volume, misleading patterns, and high processing requirements creates significant challenges in detecting and analyzing the malware accurately.

Dynamic API Resolution

One of the first operations of the loader is to start the process of dynamically resolving API calls. It will achieve this through API hashing. It will first get a module handle for `ntdll.dll`. The string for the DLL is decrypted using a rolling XOR cipher.

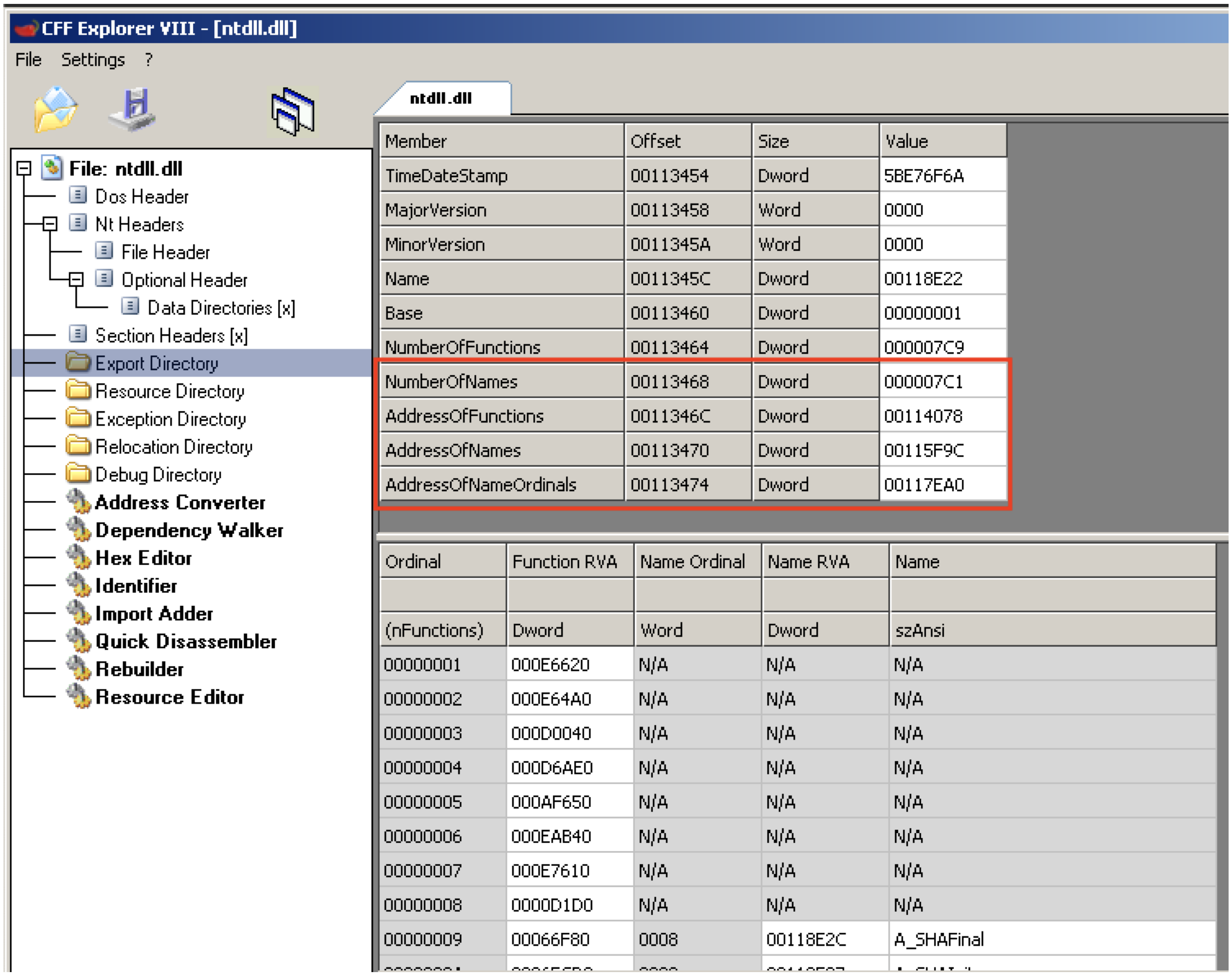


Decoding of NTDLL string

Using the returned handle, the loader will start to read the PE header of ntdll.dll and it will locate the export directory and start parsing out values that it will need to dynamically resolve the functions by hash. The loader builds up the following struct.

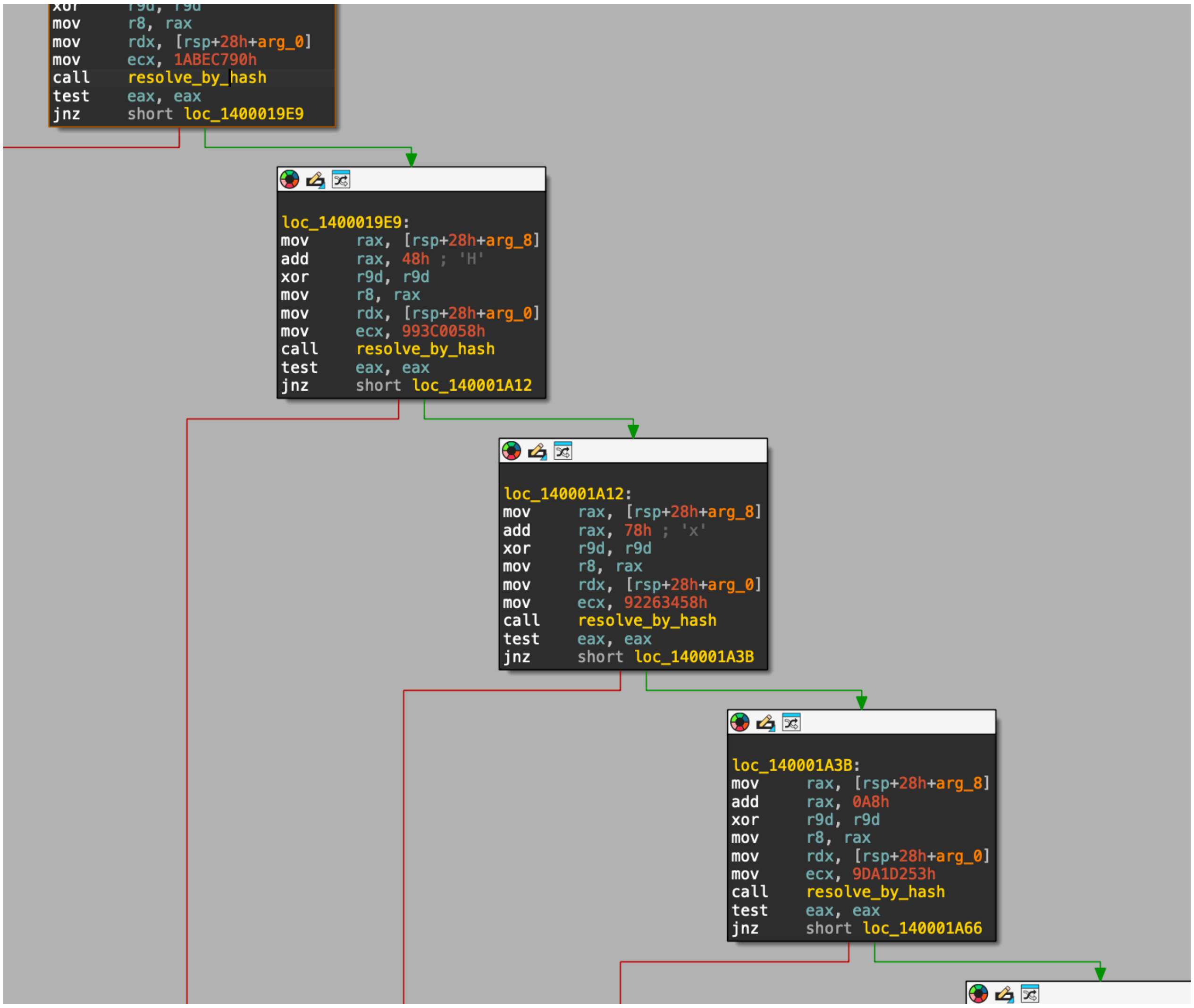
```
struct _NtDllExportInfo {
    DWORD* AddressOfFunctions;
    DWORD* AddressOfNames;
    DWORD* AddressOfNameOrdinals;
    DWORD NumberOfNames;
    HMODULE NtdllModuleHandle;
}
```

The parsed values can be seen easily from viewing the export directory in CFF explorer.



Parsed fields shown in CFF Explorer

Once the struct has been built up, it can then proceed to iterate through the export names, hashing the names to compare to hardcoded values in the binary.



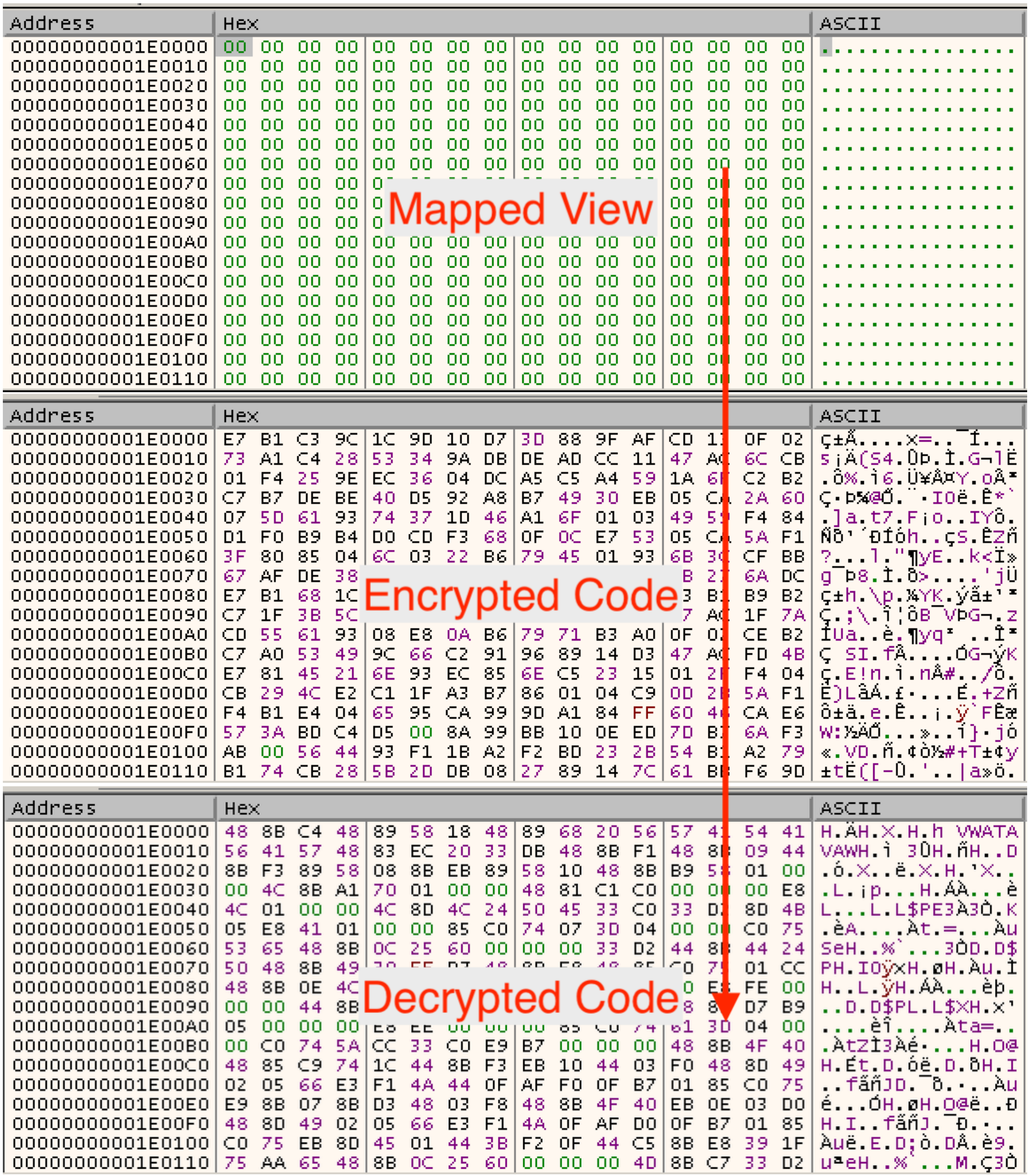
Resolution of functions by hash

The following calls are resolved, getting pointers for imports. Whilst the exports will remain the same for each build of the malware, the hashing will be unique per each build.

Hash	Call
1ABEC790	NtCreateSection
993C0058	NtMapViewOfSection
92263458	NtUnmapViewOfSection
9DA1D253	NtClose
6AF3F390	NTQuerySystemInformation
0A96AB0E4	RtlAllocateHeap
8A21A480	RtlFreeHeap

Shellcode Loading and Payload Decryption

Once the loader resolves pointers for the imports, it first calls `NtCreateSection`, followed by `NtMapViewOfSection`. This approach allows the malware to allocate and manage memory outside the standard process space. The decryption process begins with the loader rearranging the randomly stored encrypted chunks of the payload into their correct order within the mapped memory, before proceeding to decrypt each block.



Decryption stages

Before calling the decrypted code, the loader will perform one of a number of anti sandboxing checks.

AntiSandboxing/Analysis

DirectX DLL

One of the anti-sandboxing checks involves checking the installed graphics adapters to see if it is running in a sandboxed environment or not. This is achieved by importing the DLL `dxgi.dll`. The DLL is the DirectX Graphics Infrastructure library and is a core Windows DLL that provides functionality for interfacing with graphics hardware.

The exported function CreateDXGIFactory is called giving the loader a IDXGIFactory object. This allows the loader to enumerate information from the installed graphics adapters by calling `EnumAdapters`, followed by `GetDesc` from the `IDXGIAdapter` object to give a `DXGI_ADAPTER_DESC` struct.

```
typedef struct DXGI_ADAPTER_DESC
    {
    WCHAR Description[ 128 ];
    UINT VendorId;
    UINT DeviceId;
    UINT SubSysId;
    UINT Revision;
    SIZE_T DedicatedVideoMemory;
    SIZE_T DedicatedSystemMemory;
    SIZE_T SharedSystemMemory;
    LUID AdapterLuid;
    } DXGI_ADAPTER_DESC;
```

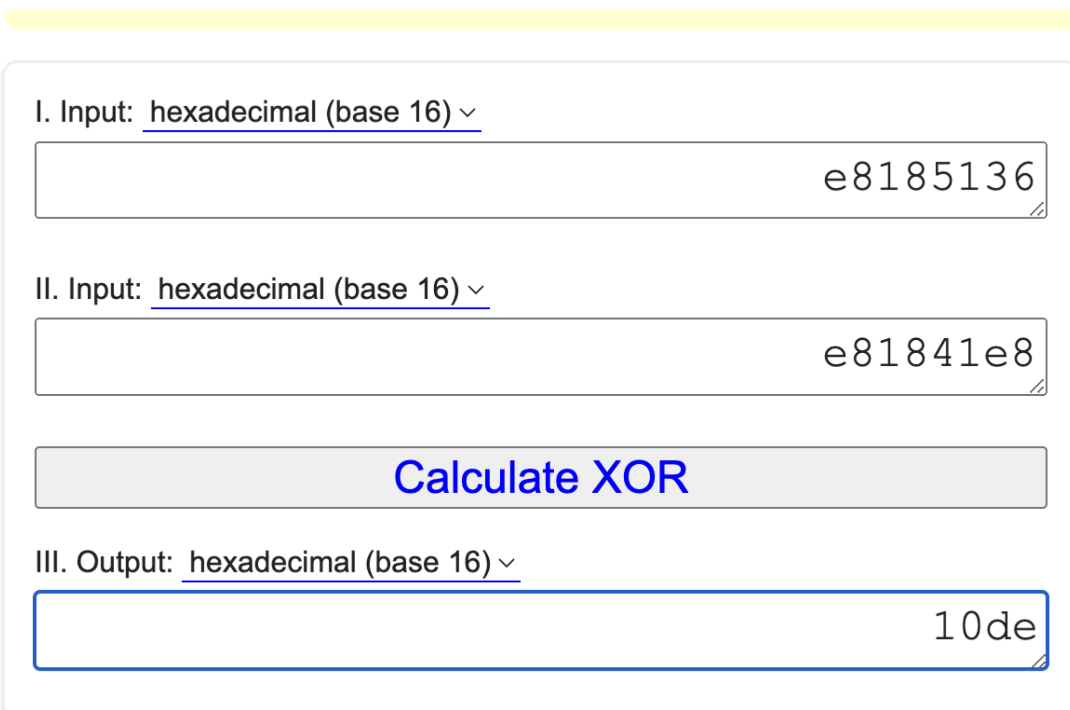
From these structs is parsed the `VendorId`, and it is compared against three values that form a vendor whitelist.

ID	Vendor
8086	Intel
10DE	Nvidia
1002	AMD

This anti-sandboxing technique has been observed in previous malwares, namely Furtim in 2016 and Invalid Printer Loader in 2023. BabbleLoader takes additional measures to hide the vendor ID numbers through using a simple XOR key and a few assembly instructions. The instructions are separated by a large amount of junk code so as to hide the values when statically analyzing the sample in a disassembler.

```
mov     [rsp+1FF48h+nvidiaId], 0E8185136h
...
//Junk
...
mmov     eax, [rsp+1FF48h+nvidiaId]
xor     eax, 0E81841E8h
mov     [rsp+1FF48h+nvidiaId], eax
...
//Junk
...
mov     eax, [rsp+1FF48h+nvidiaId]
cmp     [rsp+1FF48h+vendorId], eax
```

The decoded value (Nvidia Vendor ID) is shown below:



XOR to derive VendorID

VDLL Function

Another form of anti-sandboxing comes in the form of a VDLL check to combat Windows Defender’s Antivirus Emulator. To start this check, BabbleLoader, in a similar manner to how it deobfuscates strings to dynamically resolve functions, will decode two DLLs with exports.

The first check is to get `kernel32.dll` and look for the proc address for `MpSwitchToNextThread_WithCheck`. The second check is `ntdll.dll` with the export of `MpDispatchException`.

Call of emulated function

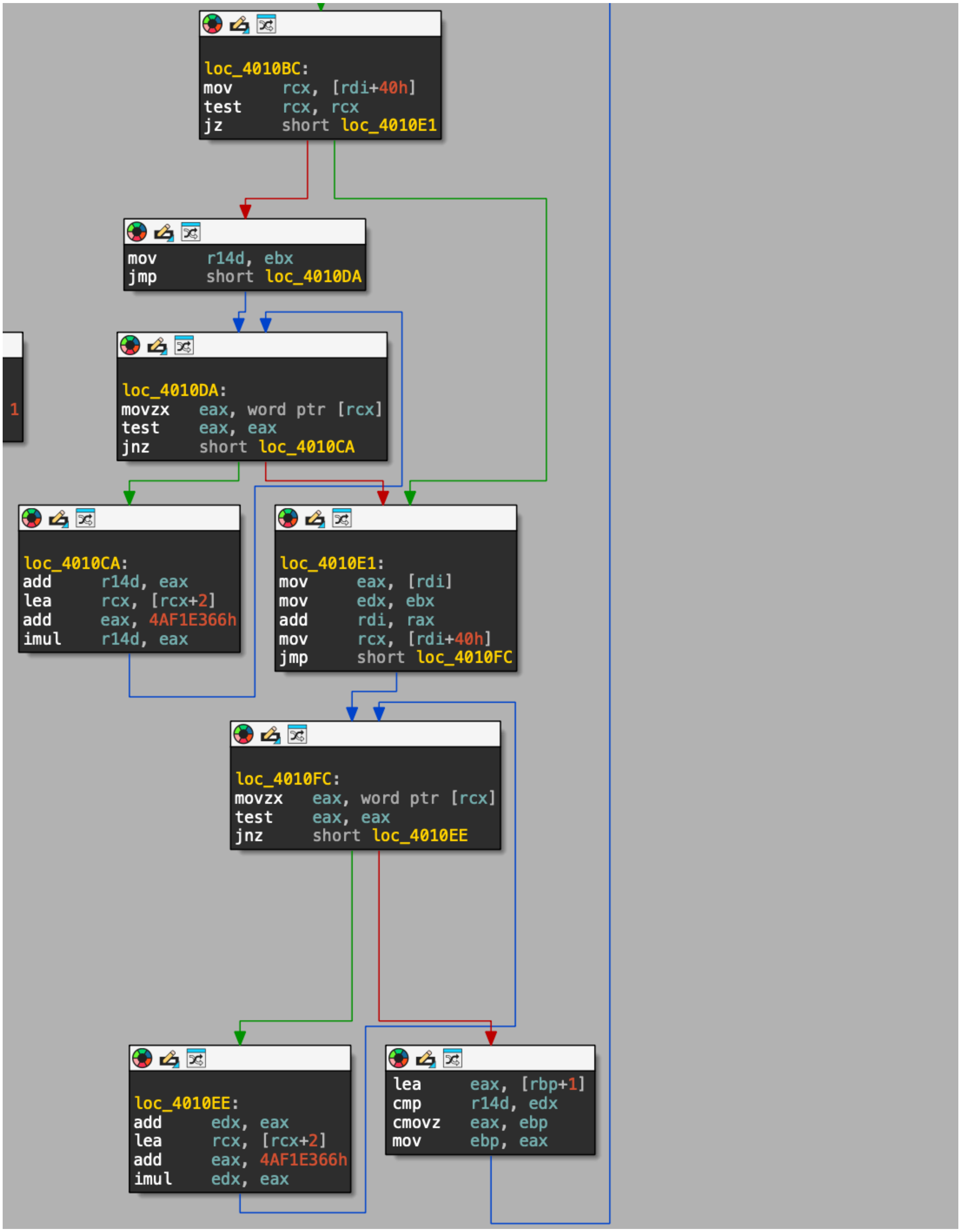
If any of the `GetProcAddress` calls are successful, it will set a variable for the loader to exit later. A successful import of any of these calls will indicate that the loader is being emulated by Windows Defender. This is because these exports only exist in VDLLs, which are modified Windows system DLLs available only in the emulator for Defender. This technique has been used by Raspberry Robin previously, and suggests that the loader developer is able to incorporate new technical research around antivirus and sandboxing internals.

Unique process count

When the shellcode payload that is stored in the mapped memory of the newly created section is executed, it performs another anti sandboxing check, this time based on the running processes in the machine.

This is achieved first by calling `NtQuerySystemInformation`, previously dynamically resolved from `ntdll.dll`. Getting the `SystemProcessInformation` class. This returns an array of `SYSTEM_PROCESS_INFORMATION` structures, one for each process running in the system.

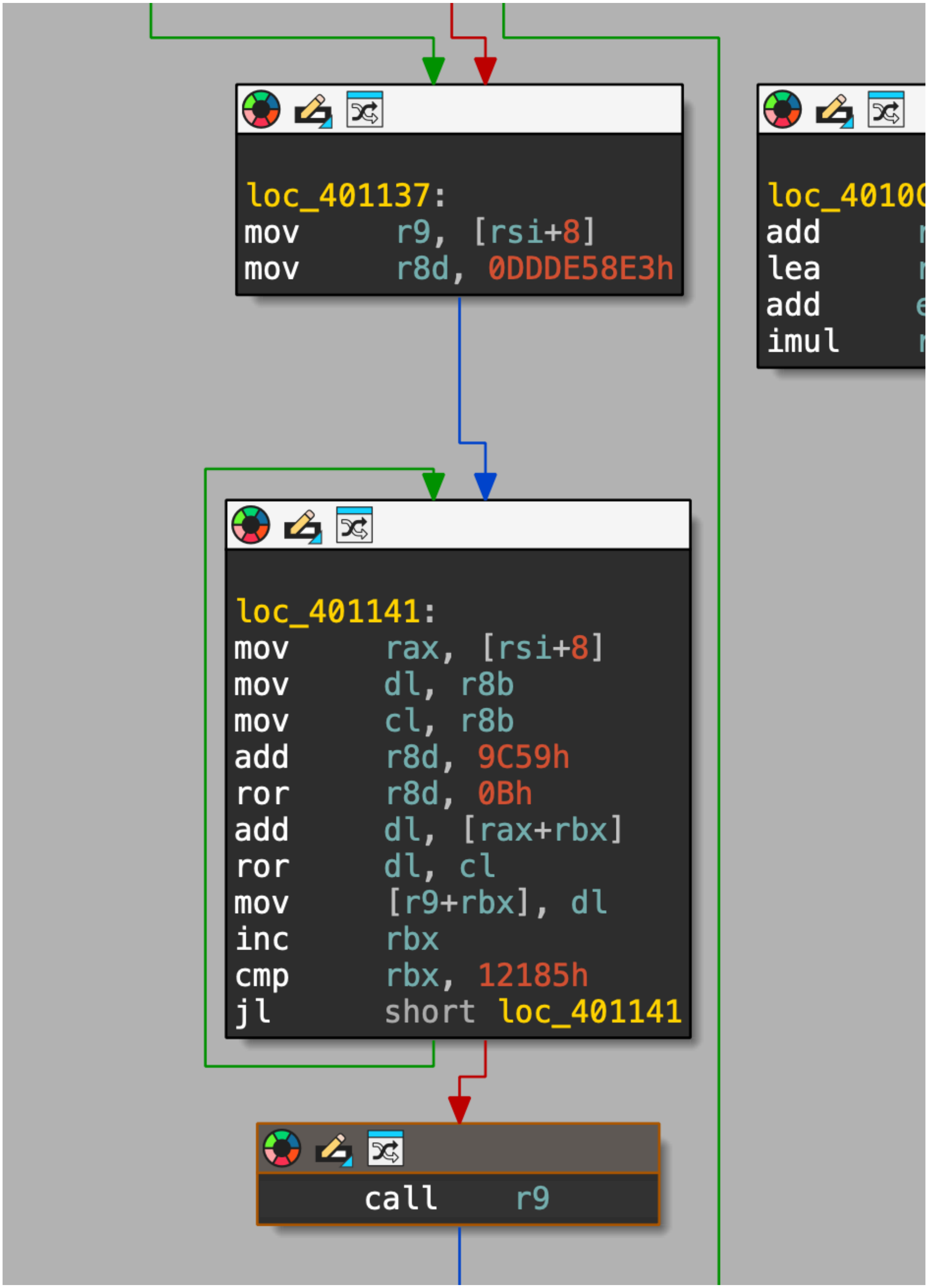
The process name for each process in the array is gathered and hashed as a checksum, and compared with the hash of the name of the process next in the array. A counter is incremented with each iteration, but if the checksums match, the counter is reduced by one. Giving the number of processes with unique names running.



Checksum counter

The counter is compared to check if there are at least 85 unique processes running on the machine. With the assumption that a true infected computer would have more running processes rather than a sandbox or emulator that is trying to be as lightweight as possible to reduce noise and costs. This technique has been employed by other malware also, such as Latrodectus.

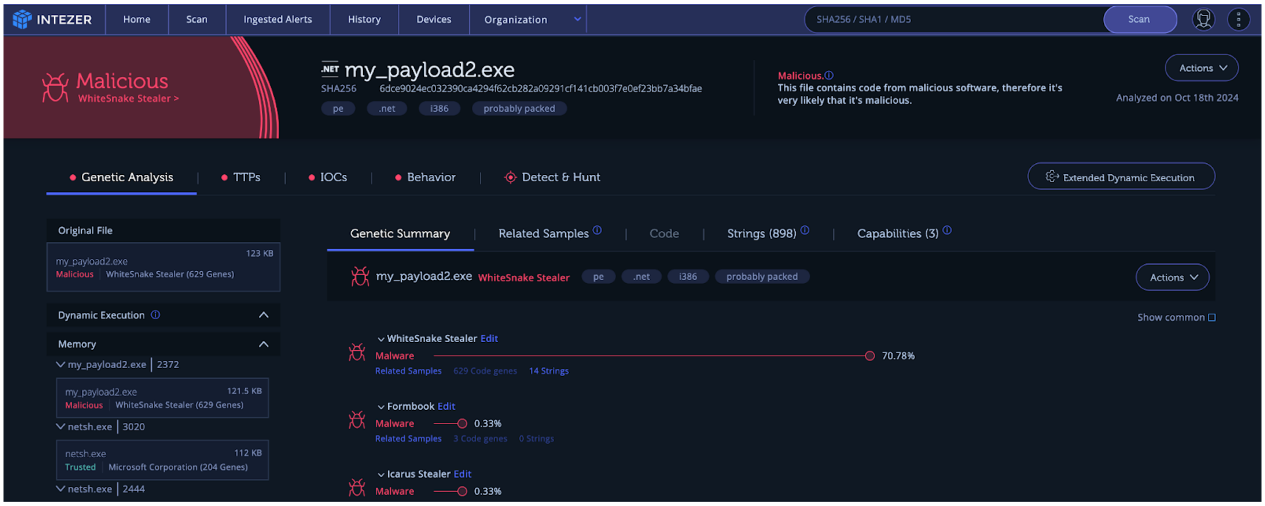
When the check has passed, the next stage of the payload will be decoded and executed.



Second stage of decryption

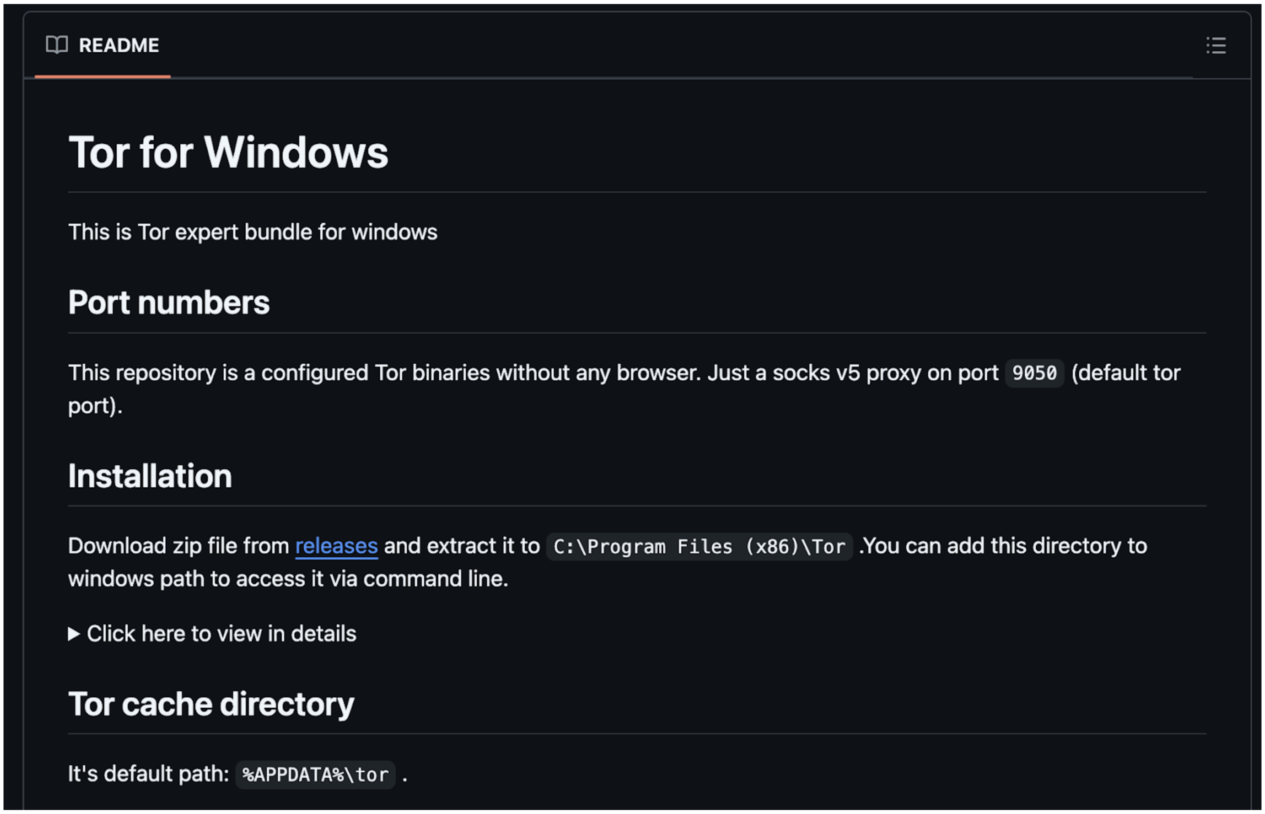
Donut Loader and Payload

The next stage in this chain is a Donut loader. This is used to unpack and execute the final payload in memory. Donut loaders have been used by many different malware and threat groups in their operations. The payload in this sample is a WhiteSnake stealer.



WhiteSnake Payload

This payload has a very interesting method of communication with its Command and Control (C2) server over TOR. The C2 communication is described in further detail in a blog from JFrog in 2023, but instead of downloading from the official TOR Project website. The payload is downloaded from this github repository.



README

Tor for Windows

This is Tor expert bundle for windows

Port numbers

This repository is a configured Tor binaries without any browser. Just a socks v5 proxy on port `9050` (default tor port).

Installation

Download zip file from releases and extract it to `C:\Program Files (x86)\Tor` .You can add this directory to windows path to access it via command line.

▶ Click here to view in details

Tor cache directory

It's default path: `%APPDATA%\tor` .

Open source project downloaded by WhiteSnake

In other samples, Meduza stealer has also been observed. There may be other stealer payloads delivered that have not been observed yet.

Considerations for Defense

The use of loaders is a long-standing technique incorporated by threat actors. In order for modern day threat actors to have any success against the many layers of detection employed by security vendors, they too must incorporate multiple layers of defense within their own builds. It is a never ending arms race between attacker and defender. Each side imposing increasing costs on the other in a frantic effort to come out on top, no matter how short that time period may be.

The better that the loaders can protect the ultimate payloads, the less resources threat actors will need to expend in order to rotate burned infrastructure. BabbleLoader takes measures to protect against as many forms of detection that it can, in order to compete in a crowded loader/crypter market. The types of protection utilized protect the loader against hash, rule, genetic, static, dynamic, and AI forms of detection, imposing costs upon security vendors in the hope that the cost of detection will be so high that it will cause security vendors to overlook analyzing these files.

The developer behind this loader demonstrates an active engagement with current security research, rapidly integrating new techniques to enhance evasion capabilities. For instance, recent anti-sandboxing features reflect insights from research on Windows Defender presented by white-hat experts at Black Hat, allowing the loader to better evade detection by Microsoft's defenses. This adaptability shows a strategic commitment to keeping the loader ahead of evolving security tools by adopting the latest innovations in bypass techniques, making it more resilient and harder to detect with each new build.

Many security vendors will look at using AI to help in future cases with combating these loaders. The loaders of the future are already well equipped in this fight. The loader's layered obfuscation tactics pose a formidable challenge for AI-based defenses. These techniques flood the AI with irrelevant tokens and misleading patterns, making it difficult to distinguish meaningful actions from noise. Each layer of complexity forces the AI to process massive amounts of data, drastically increasing computational and financial costs. By weaponizing this volume and variability, the loader effectively undermines AI's pattern recognition and analysis capabilities, pushing the limits of automated detection systems.

There is no signs of slowing down in the pace in the thriving loader market.

IOCs

BabbleLoader:

052c776fdc9700dfb37f964a73d461a57efad30a01bcf54505d7abcd601e6ff3

0ad8513b62a778d7e426627be3ed2dbaf00d99b9802a1f566dc9203e3d311fc3

0f6847d33cb38b0ed6dc1d8cfe3dc5d2e293d91c4880e3b4f5ddb77fd9d4cd1f

114b868f319162c5d6ff92796e41910f54de0e89f895a066fd4980c6dba2e323

1367fb270f35512b17fe5e73cc0233ace272daafe15cf94e45f696431f52332f

1537965c7722a9886d542688fcbafecd1248b2fbd56e9a90a803a50e880e1bb8

16200bbe4646fe8cefeee5be20ce55c50300485f3356ab181fb930bd02536709

1da4de2b4b87bff7f9f1a3208c5c663a06f7f9b67f918e5a5e8e860e759b7075

200289d5a408a2e49a894228edb3324548ca5c5c0283d09486aa287df41f15bc

217d7501287ae894f47bd04253bd184d1901dd131b0cd15bcbbeba5158049d5d

22866e6ded40916de8002606f82e44ee141f27c3340fa2c4d16514624ee05a72

237812322bbbcf47feeb79b8e91b97d00453ffd5deb52c819c183b45d18b0b5a

25923b822e9a1374817caf79375170b944f2762b1e3f2add921008ffec702740

2a8a340fc9c395fe23211ac95d124b64452d49c67b069f53aaf3dbe16e95791d

2b6bff7b8c23f1fa526e116c7577c32845d5b969c68a66850c305a351adc9726

2d6b50003436ed489d1b46566eb879e317e1b9a5b6edb12f3cbb4c8a8d932a08

2eab850166944175e5fac4c89706328a58dcef55dbc22ff20342d1d246ba76b9

2ee32c46207119f6851f2869203124c104c72cfdf9622416252ae3405f485cd2

328d92b71034d74c016b1f8e70217be3f432a2ade8fe44522f84980fd0dbb1f9

33e42e7828cda7987d17342e0eb8134f590cd3d291dbc75f13334259a4908ba1

3bf5f07059a24fb803c6fefb874f000e9c300372b1b870e48b4935bd0219fe2b

401209ec9dda222984fd5cb775a6b6c2e651d88c04a506c9058cb1decdce869d

451e1bec8476a89c7d2b526071fa2918187f2f5b3ba9362e6999114993a74da5

455cd0db2de92ee348295780f8fc7a32a5406a5986a4d162761680f11b6346b1

466a8af8d0b51ed82aec35b17b845e6baf77ada259aa2fd5591024a01d8e31b5

46b355d25b95d7f9d7029f1ee1a346028e3c5bdec9e6c9245c12d1607cb1c686

46f0e190cd346d1eb6d0c27386bb3aceebf4ad44b25d253cac4063e2ccde9028

478eb22a1f1be2ef6e70625cf42ca61c716389135acbb705c0e21f0cf330bf46

47a71eb078b14a92eb5fb990f606aa48e535860af90ebc5e075c8b2e4d829633

4b7fa864007357e3e799eeb4a963042565217855 1a9c37181fc6ea86660af814

4ba95478ea0ac78e038d30693fabf95244bd70e40b36b2a928f3854551d6fa78

4bed4960a896ebeafa9a9421d7ecf389205a2f0216ad911bdcd80f28237159e6

4e40aaddf718b70f397d449f8ca9a577ef0106f281ccb50f0b5cde531b758881

5305556bee271232973a9e09c4776a3b386964112785db638b225b2cc61d9af7

53e451750c099f33f80a3652d9f2a804390de0f867af13ae22ad0fbf4b15f8c3

5493fa6f2ab69da66352532b2c13e7e461bcde6cc2810a6f6af88e139dde1ae7

5665c96975c959b5e8057b7aed46f7c203335aefa35f5e290c538e9300e3e293

5b9481d9022b0efcaed04513d338048de4aa3e1328bacc0966486ef322c0d086

5eb3bb67656d990ceec07f55c78dcd8032a7cf00ac919a399e3642b177f68381

60ecef2d0a966db913bff15872c072175b895e16271351c43e5a0cf9e4018f22

643dde3f461907a94f145b3cd8fe37dbad63aec85a4e5ed759fe843b9214a8d2

69ad389722dd8b49590b2aa014f703b39737894073c7339ea44c94d296e00273

78f6c822cee2b0587df145d67478cce5bbeb76147a7846d08b7b6fd09aa36ce2

796c245c5bb1e7c1dcd52c4e8f83e1c707e391f6409ee9b5e1dd18658ff0e05f

7df313618a02e8e9961ddb1c3289956eb18361f1ca9fb639d64a00fae7568a4b

7e5ba9e3ccc1cb52d24c21c6d378a32bb540a8519789d8cf796e5dd351fc6958

8907a8454ef56d64bf788b9c8c64bbaaf187be7a9666d8d8331fd187c49c6031

8a28e457b19060678d5d007b291722e1dea92f69249e1588ca5b97eb1fe10807

8cc2e1104480875ee237bf4ca9f3d83e46ca213f5c88df95be0d78e05c7c2d71

8cf8bea6842219e565720919372e4aa530942b28d533231043ee57e7bb424cbe

8d8c3b6be212ce645566311ce95ad9ad3aad37795042882adefda9716deb2eab

8e63b1f7f8e29b9a714f796e2e8ca0cd1094086e2d0a5de21601e23e1792a906

9125c13250a14905a4fd97978a3a6dbba80df01e73d98f8d4fa2d19b49d9fda0

9314ea889f93da5cd39129840a42bd5f228538686a2345f56e757e5a5d956dee

9bf7a01254fed809e0f564f28a3cf54156ea98f85d3b633ae3a213a87f9db143

9dfb8ed499b667d782ae3a4ce40472893a789ed973f48884b47358536b6a76e8

9fa7574f35fae3d309c8cefe0e8a43d07afb6cefaee0caa3b2538263bd4a7ec5

a01ac4244102e3958296c70d71e3d951f11abcc355458d1918d081587b151d90

a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87

a29e108e912c77ed565873bd205da01ed0e6001d18b442139c06827428d2eba1

a3b45619606d4c3c487047786e3d51a98fdcc1fdc43dc7b6f6e80974fd0a9c97

a695cb493631962a4c2fd61a094cb0b952ce708a99af714772cddd4991f32df0

ae6ee6bf2f9890ed83922e5c80770dd88faa9b32b2211462ea2eed29bf1bf6c5

b14af38c4230de20c7c4fefc1e3c5fffb1562bacedfebc56a508f55182a6fe88

b1ebe1794e091fd82a34d6806f18f64ebadb5d3b2343a661c481fb7c54cb872f

b2a91277e5fb40a0a38215142f683554b4e7c03ccd439e0d056c56b031a5bec5

b72d9ae8484b91ec9c6167e6707617f495622f3b684f6b3e30b29106891c778b

bb4f812f8bb4e7b33d7b583407370a5351d079f63b26956adc7ab317b3d90601

bdd6bd29937059dd944fb09163a24e4482c5ce420d3de749e5e46c6c25b2ea86

c2a95f22cfee1f4df67a424e30425b59c23db265bff611f2ee653d71b30a70d8 ca67f61b5f8d20ec3f45dbbfc355045a8ceee15396f1cad032850a3ee23a42b3

cd3f064d088a3a6a6ad03da148701fb6b660866b8aac2a808359505620166641

d7967661947ca835deddec30ae6e62d580718cbdeafb42cd6d0f038a407edcf0

d9cea34db0d1dc016dd4007d8cd11416f095c41b0639f13af1eb6ad675651df2

db282cae419ed5af3338f65f170ecd7b312cac2500b5cb2c8824721ba981c361

db41e032193becc097d0da85cac74cf1f519b85cf731f783ccea11c1e20ad23f

e09c36993e1c29b6ef0f1c73e02aee54686c0df49b6d87b577e70f261313acaf

e13f20752f6298728ac0463a3f4b0657d5657ca7710e63a27ac1179078ac71f6

e1448680114cb3dd06aa81a3b1037f77e6d5b3f6dce213aa38cffdec72d59e74

f2f23a963952c1a822484382bf4c68cd8b7278400ad2d8bacf3235ba2fc42a89

fa292bfcf81223bab0f79d4ce08187e37d68960005629df0241ea22f0b95d7a8

fc589aa3529a057fee52a1c9bd9bb19fa42bbfd291b7dbb3791e77eced376640

ffcae0093d509563b47b1d0cef3aa455a4358de3a1d2c2b84c298a927aa238e8

WhiteSnake Stealer:

6dce9024ec032390ca4294f62cb282a09291cf141cb003f7e0ef23bb7a34bfae

Ryan Robinson

Ryan is a security researcher analyzing malware and scripts. Formerly, he was a researcher on Anomali's Threat Research Team.

| Terms Of Use | Privacy Policy