Teasing the Secrets From Threat Actors: Malware Configuration Parsing at Scale

unit42.paloaltonetworks.com/teasing-secrets-malware-configuration-parsing

Mark Lim, Daniel Raygoza, Bob Jung May 3, 2023

By Mark Lim, Daniel Raygoza and Bob Jung

May 3, 2023 at 6:00 AM

Category: Malware

Tags: Advanced WildFire, IcedID, memory detection, WildFire

This post is also available in: 日本語 (Japanese)

Executive Summary

Configuration data that changes across each instance of deployed malware can be a gold mine of information about what the bad guys are up to. The problem is that configuration data in malware is usually difficult to parse statically from the file, by design. Malware authors know the intelligence value as they provide directives for how the malware should behave.

Malware is like most complex software systems in that there are many advantages for code reuse and abstraction. Therefore, it is not surprising to see that the concept of software configuration is pervasive across the various malware families we analyze. After all, it's pretty hard to imagine a stereotypical cybercriminal wanting to bother with recompiling their code to change an IP address or whatever else, when going after different targets.

But the good news is that statically armored configuration data can often easily be found and parsed directly from memory. We will cover a nice example of an IcedID (information stealer) configuration, how it was obfuscated and how we've extracted it.

Palo Alto Networks customers receive improved detection for the evasions discussed in this blog through Advanced WildFire. As we continue to parse and extract this information from malware families at scale, we hope to build out a pool of threat intelligence that will better help us understand the campaigns and tactics of the various threat actors who are targeting various organizations.

Related Unit 42 Topics **Memory Detection**, **Malware**

What Are Malware Configurations?

So what exactly do we mean by the term "configuration" when talking about malware? Outside the context of malware, we think of configuration in terms of defining how systems should behave. For example, we would consider the rules used to define which networking routes for a firewall are allowed, or which font size your web browser uses while you read this, as configurable information.

For malware, this is no different. Malware configurations are just collections of elements that define how a malware operates, such as the following:

- Command-and-control (C2) network addresses
- Passwords for remote administrators

- File paths in which to drop persistent payloads

The way these elements are embedded in malware components tends to be specific to each malware family. Also, they might evolve over time as malware undergoes development, or when malware authors change their build process.

Generally speaking, malware configuration elements tend to be the properties of malware that the authors want to make easily editable between campaigns and deployments without requiring manual code edits for each one. Malware configuration elements can also expose latent behaviors and malware infrastructure that are not typically observable under routine dynamic analysis.

Malware configurations have intelligence value for security practitioners because they provide insights into campaigns over time. In some cases, defenders could use them as actionable artifacts for network detection, or for identifying infected hosts. The successful extraction and validation of a malware configuration can also be used to reinforce our confidence when identifying a file as malicious.

Because malware configurations have value to security systems and defenders alike, it is state-of-practice for modern malware authors to protect their configuration elements using different techniques. These protections often include a blend of encryption, obfuscation and compression. They might also be layered with evasive techniques.

This protection poses a significant challenge for malware configuration extractors that operate solely by using static analysis, because all of these protections must be detected and bypassed before extraction can be performed. Using an advanced dynamic analysis sandbox combined with intelligent runtime memory analysis makes it possible to bypass many of these protections and pinpoint the best opportunities to perform extraction.

When we represent and store these configurations using standardized schemas, it enables us to extract maximum value through automation, machine learning and interactive analysis. The DC3-MWCP library defines a schema for many of the most common configuration element types, and it provides a simple library for serialization to JSON.

The MITRE MAEC and STIX projects also provide us with a more general vocabulary for representing malware configuration elements. This also allows us to correlate the elements with observable objects collected during dynamic analysis.

IcedID Analysis

Let’s look at one IcedID binary and how its configurations are encrypted.

Hash	05a3a84096bcdc2a5cf87d07ede96aff7fd5037679f9585fee9a227c0d9cbf51

This particular attack chain, shown in Figure 1, was discovered in early November 2022. It delivered IcedID, an information stealer also known as Bokbot, as the final payload. This threat is well-known malware that has been attacking people since 2019.

The following diagram shows the infection chain.

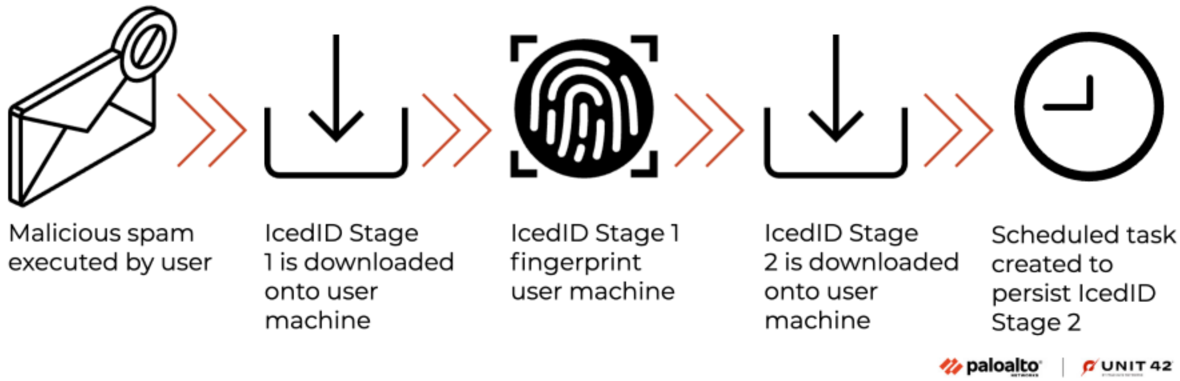

Figure 1. IcedID infection chain.

Authors of IcedID took pains to hide their configurations. Recent samples of IcedID stage two would only be downloaded if the victim's machine matched the requirements of the threat actor.

The configurations of IcedID consisted of C2 URLs and their campaign IDs. The C2 URLs included some that might not be revealed during the execution of the IcedID binaries. The campaign ID links IcedID samples back to specific threat actors.

We will go through the following steps to extract the configurations found in the IcedID stage one and two binaries:

1. Unpack the IcedID binary
2. Locate the encrypted configuration data blob
3. Extract the encryption key
4. Decrypt the configuration data blob with the encryption key

Unpacking IcedID Stage One

IcedID stage one unpacks itself by first allocating memory using the VirtualAlloc function. This is followed by erasing the allocated memory using the Memset function, as shown in Figure 2. Finally, it copies the unpacked data to the allocated memory using the Memmove function.

To dump the unpacked data, we set a breakpoint at Memmove. The second argument of Memmove contains the address of the unpacked data. Figure 2 also shows the DOS MZ header of the unpacked IcedID stage one in the right-hand side of the hex dump.

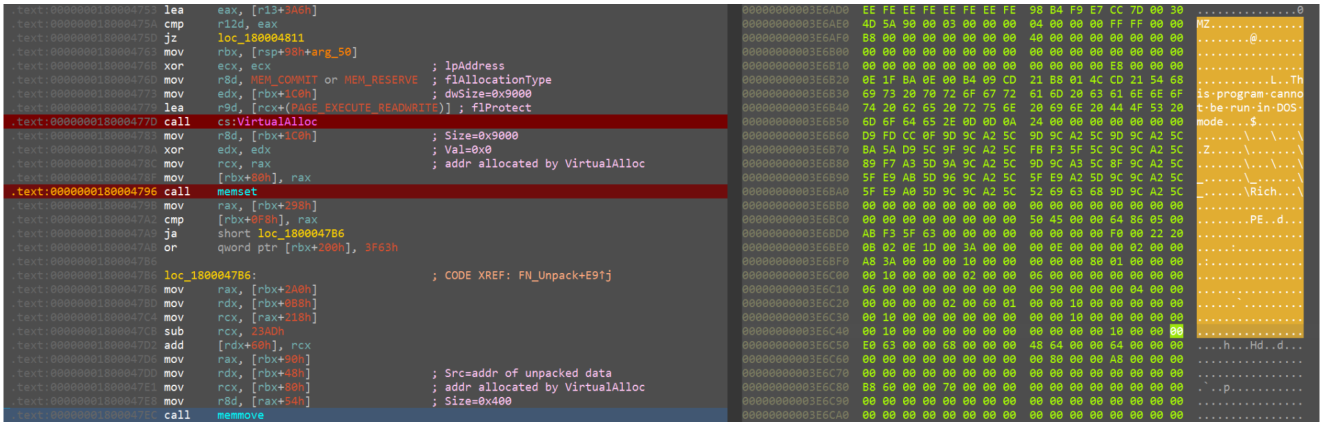

Figure 2. Unpacking IcedID stage one.

Locating the Encrypted Configuration Data Blob

Next, we located the encrypted configuration data blob using the unpacked stage one IcedID. While debugging the unpacked IcedID stage one file, we set a breakpoint at the address that called WinHttpConnect, as shown in Figure 3. The address pointed to by register RDI contains the string of the C2 URL.

Figure 3. Debugging IcedID stage one.

By backtracing the code, we located a function that used the decrypted configuration as shown in Figure 4.

Figure 4. Tracing code in IcedID stage one.

Tracing the code flow back, we found the loop that decrypted the configuration, as shown in Figure 5.

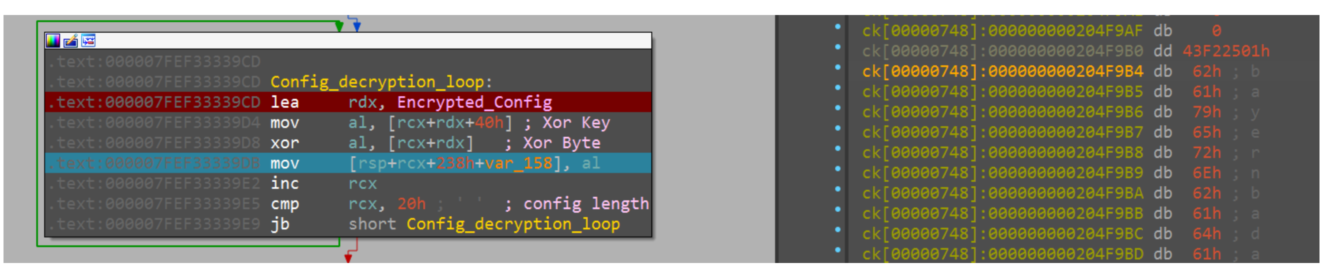

Figure 5. Configuration decryption loop for IcedID stage one.

The instruction at 0x7FEF33339CD loaded the address of the encrypted configuration data blob (Encrypted_Config) into register RDX.

Extracting the Encryption Key

The instruction at 0x7FEF33339D4 reads the encryption key. The key is 0x40 bytes offset from the address of Encrypted_Config. We also learned the configuration is 0x20 bytes long. An XOR loop was used to decrypt the configuration.

Decrypting the Configuration Data Blob With the Encryption Key

After gathering the encryption key, the encrypted data blob and the decryption routine, we can now decrypt the configuration using the following script shown in Figure 6.

```
from struct import *

enc_config_blob ="3a415bc8cb53f146a2b969d00ce010bc20ba588dca4cb27778b17acf8e339c71f7607a9dd
bytes_enc_config_blob = bytes.fromhex(enc_config_blob)

key_offset = 0x40
config_len = 0x20
bytes_enc_config = bytes_enc_config_blob[:config_len]
bytes_key = bytes_enc_config_blob[key_offset:key_offset+config_len]
bytes_clr_config = []

for x in range(config_len):
    byte_clr = bytes_enc_config[x] ^ bytes_key[x]
    bytes_clr_config.append(byte_clr)
bytes_clr_config = bytes(bytes_clr_config)

Campaign_ID = unpack('I',bytes_clr_config[0:4])
C2_url = (bytes_clr_config[4:]).decode('utf-8')

print(f"Campaign ID :{Campaign_ID[0]}")
print(f"C2 url :{(C2_url)}")
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
PowerShell 7.1.3
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

PS C:\Users\RE> & "C:/Program Files (x86)/Python38/python.exe" c:/Users/RE/samples/icedid_ida/Config
_decoder.py
Campaign ID :1139942657
C2 url :bayernbadabum.com
PS C:\Users\RE>
```

Figure 6. Configuration decryption script for IcedID stage one.

The decrypted IcedID stage 1 configuration has the following format, as shown in Figure 7.

```
struct IcedID_installer_config
{
    DWORD Campaign_ID;
    unsigned char C2_url[0x1c];
}
```

Figure 7. IcedID stage one configuration format.

From the decrypted configuration, we can extract the following IoCs:

C2 URL	bayernbadabum[.]com
Campaign ID	1139942657

Now, we will decrypt the configuration for the IcedID stage two binary.

Unpacking the IcedID Stage Two Binary

As the IcedID stage two binary uses the same packer as stage one, we will not repeat the unpacking steps here.

Locating the Encrypted Configuration Data Blob

We set a breakpoint at the address that calls Winhttpconnect, as shown in Figure 8.

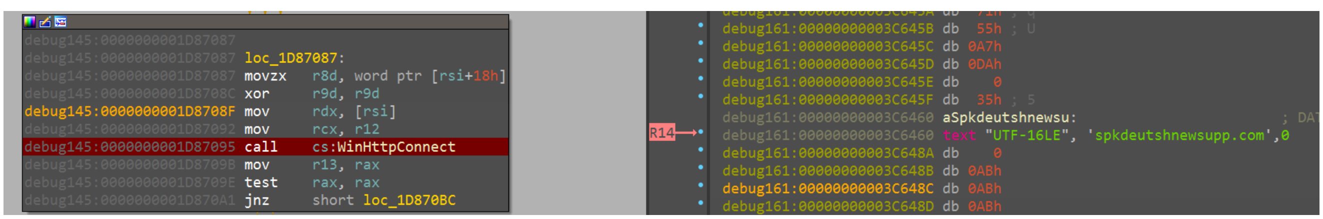

Figure 8. Debugging IcedID stage two.

After tracing the code, we located the function that used the decrypted configuration, as shown in Figure 9.

Figure 9. Tracing code in IcedID stage two.

Extracting the Encryption Key

Tracing the code flow even further back, we found the function that decrypts the configuration. The first few instructions located the encrypted configuration blob. The encrypted blob is 0x25c bytes long. The encryption key is the last 0x10 bytes of the encrypted configuration blob, as shown in Figure 10.

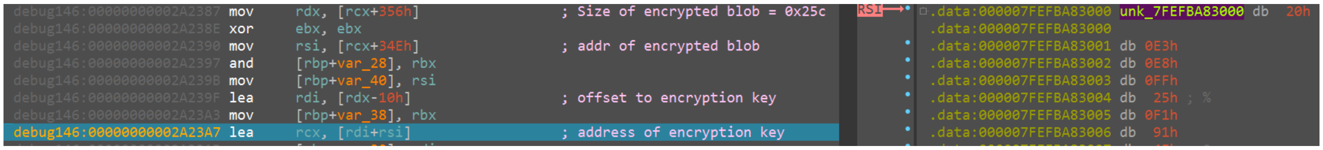

Figure 10. Loading the encryption key for IcedID stage two.

After retrieving the encryption key, the next step is the loop to decrypt the encrypted blob, as shown in Figure 11.

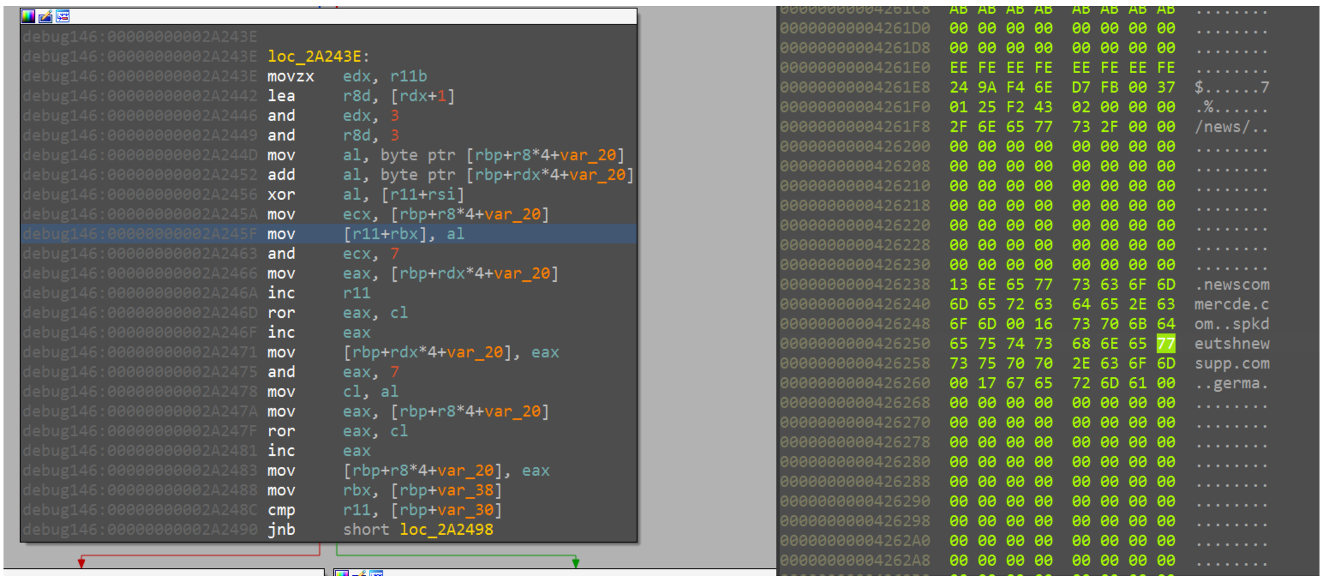

Figure 11. Configuration decryption loop for IcedID stage two.

Decrypting the Configuration Data Blob With the Encryption Key

We replicated the instructions in the decryption loop using Python. After gathering the encryption key, encrypted data blob and the decryption routine, we can now decrypt the configuration using the following script (shown in Figure 12).

```python
def rotate_key(key, x, y):
    temp_key = bytearray()
    temp_value = key[y:y + 4]
    temp_value = struct.unpack("I",temp_value)[0]

    rotate_value = (temp_value & 7) & 0xFF
    temp_value = key[x:x + 4]
    temp_value = struct.unpack("I",temp_value)[0]
    temp_value = ror(temp_value, rotate_value, 32)
    temp_value += 1
    temp_value_X = struct.pack("I",temp_value)

    rotate_value = (temp_value & 7) & 0xFF
    temp_value = key[y:y + 4]
    temp_value = struct.unpack("I",temp_value)[0]
    temp_value = ror(temp_value, rotate_value, 32)
    temp_value += 1
    temp_value_Y = struct.pack("I",temp_value)

    temp_key = key[:x] + temp_value_X + key[x + 4:]
    temp_key = temp_key[:y] + temp_value_Y + temp_key[y + 4:]

    return temp_key

def decrypt(data, size, key):
    outList = bytearray()

    for i in range(size):
        x = (i & 3)
        y = ((i + 1) & 3)

        c = key[y * 4] + key[x * 4]
        c = (c ^ data[i]) & 0xFF

        outList = outList + struct.pack("B",c)
        key = rotate_key(key, x * 4, y * 4)

    return outList
```

```
OUTPUT    TERMINAL    DEBUG CONSOLE    PROBLEMS

PS C:\Users\RE> & "C:/Program Files (x86)/Python38/python.exe" c:/Users/RE/samples/icedid_ida/Blog/a903/Config_decode.py
{'BuildID': 1139942657, 'uri': '/news/', 'c2_urls': ['newscommercde.com', 'spkdeutshnewsupp.com', 'germanysupportspk.com', 'nrwmarkettoys.com']}
PS C:\Users\RE> 
```

Figure 12. Configuration decryption script for IcedID stage two. Note: Jquinn147 and myrtus0x0 published a similar configuration decryption script for IcedID in May 2021, called IcedDecrypt (GitHub).

The decrypted IcedID stage two configuration has the following format, shown in Figure 13.

```c
struct IcedID_installer_config
{
    DWORD Campaign_ID;
    DWORD uri_len;
    unsigned char uri[uri_len];
    unsigned char null[];//padding of null bytes
    unsigned char C2_1_url_len[0x1];
    unsigned char C2_1_url[C2_1_url_len];
    unsigned char C2_2_url_len[0x1];
    unsigned char C2_2_url[C2_2_url_len];
    unsigned char C2_3_url_len[0x1];
    unsigned char C2_3_url[C2_3_url_len];
    unsigned char null[];//padding of null bytes
}
```

Figure 13. Configuration format for IcedID stage two.

From the decrypted configuration, we can extract the following indicators of compromise (IoCs):

C2 URLs	newscommercde[.]com spkdeutshnewsupp[.]com germanysupportspk[.]com nrwmarkettoys[.]com
C2 URI	news
Campaign ID	1139942657

We have manually decrypted the configuration for both the IcedID stage one and two binaries.

Scaling Up

Now that we've discussed the work of figuring out how to target the configuration data in memory, the next challenge is to figure out how to perform this at scale. The massive scale of most malware processing systems means that most practitioners looking to build out a configuration extraction system will need to be careful about adding additional overhead. This means that we will need a mechanism to intelligently identify only the samples of interest for each parser, so we're not unnecessarily running dozens of parsers across millions of samples.

We think a reasonable approach to this problem involves using intelligent runtime memory analysis, as it provides us with excellent visibility into the secrets malware authors want to protect. A typical workflow for our malware configuration extractors includes the following activities:

- Scanning memory and/or other dynamic analysis artifacts
- Applying a noise filter on the results to identify the best candidates for extraction
- Performing extraction using the best fitting module and storing the results for reporting and indexing

Generalizing this common workflow presented us with the opportunity to make the following improvements:

- Optimizing the search phase by only scanning analysis data once in most cases
- Applying abstractions and reusable code for many common tasks
- Limiting the impact of modules with problematic inputs or other bugs
- Giving our security researchers visibility into the performance of their modules

The following example shows some of the IoCs from a recent IcedID extractor after being deployed at scale. Having a nice framework for deploying configuration extractors means that once you are finished crafting a configuration extraction script, it’s time to kick your feet up and relax while hundreds of configurations flow into your malware configuration database.

34	87b7f4970787ed87929787c1f80efaec1a…	23967…	/audio/ agroperepravvo.best heffertopper.best cwertoposler.cyou …
35	8e24d045946252edb2fd63d83136d8264…	23967…	/audio/ agroperepravvo.best heffertopper.best cwertoposler.cyou …
36	13ad7de7f561825af82ab9ba920f82b729…	15084…	/audio/ chainoftheapril.cyou unproffesional.club
37	6966dce3e94a2451284d8dcfb801b3846…	14765…	/audio/ chinadedoing.best musiciange.club
38	884fe75824ad10d800fd85d46b54c8e45c…	15259…	/audio/ colombosuede.club colosssueded.top
39	ee0e26f57329033b24a27ff67198392ebb…	30928…	/audio/ eveningstarz.top visitgeece.space tourtogreexce.space …
40	fb3a40e249ebffa480b40c6cddb2c2b7b9…	26460…	/audio/ felpojdhf8980.cyou azoperfdeoti85.xyz
41	4015c3bdb45127f210d6e9f6b1607c804d…	26833…	/audio/ funnymemos.shop trythisshop.club shopoholics.best …
42	f44d8201ad5ca7c3a78c086935fa2d9d9c…	63706…	/audio/ gelevandren.cyou greenflopper.best qassertolik.top …
43	bc8d2e218ffa72a1788e5270167dcca9d3…	63706…	/audio/ gelevandren.cyou greenflopper.best qassertolik.top …
44	77b5b0edb6f4d4a067ec9275af9f6167a8…	41633…	/audio/ ifitislovenosad.cyou nomersimore.pw
45	1430b28b39a4f495c8a88aeb49ca5b843…	26934…	/audio/ karimorodrigo.pw airtopolos.best
46	8c739e65dc852000ada649701d0996174…	26934…	/audio/ karimorodrigo.pw airtopolos.best
47	bdbc3850d100b517146a20b896e65eb2…	26934…	/audio/ karimorodrigo.pw airtopolos.best
48	db74e599d75da93640754f39f6795950a…	26934…	/audio/ karimorodrigo.pw airtopolos.best
49	6ac0970d4b2a3ff0a279f1632c28c31f2f3…	15622…	/audio/ maseratipirosh.top tyrek87.cyou
50	c0ebb6d2b3647426b5b712c0ab956f8f85…	15622…	/audio/ maseratipirosh.top tyrek87.cyou
51	40c60fa13696155e04dab4d6086d32c3b…	25826…	/audio/ pashamasha.top pohindra.online
52	b83b84fc4d0cee9ab6a9c39246ae46d79…	25826…	/audio/ pashamasha.top pohindra.online
53	15f9a0d1de7639255ce230e648ae2254e…	25826…	/audio/ pashamasha.top pohindra.online
54	0728a76febf93a8bf5b5edc9335655f93f4…	21850…	/audio/ revopilte3.club aweragiprooslk.cyou
55	20fbdedfeb0334ad02265234f4defe6e43…	21850…	/audio/ revopilte3.club aweragiprooslk.cyou
56	c7a41aaae47af9ebc6bcabb267e1d11d9…	21850…	/audio/ revopilte3.club aweragiprooslk.cyou
57	f0ad9320f60ef590cee3e78900264c7099…	21850…	/audio/ revopilte3.club aweragiprooslk.cyou
58	0f5a33610c5449b4aba2aedd5fa2e6833b…	13494…	/audio/ sadammanopore.cyou everyonemustbe.pw daskurilla.pw …
59	c90020154188cc9bf10812b623ae2d063…	13494…	/audio/ sadammanopore.cyou everyonemustbe.pw daskurilla.pw …
60	e0171caf630b9e1d6d57f18699db78bfc4…	13494…	/audio/ sadammanopore.cyou everyonemustbe.pw daskurilla.pw …
61	e7bea91d8b15c7d6aa87857fa4062e863…	13494…	/audio/ sadammanopore.cyou everyonemustbe.pw daskurilla.pw …
62	a09d8c487a135b973af532247d62f46695…	26148…	/audio/ timerdisclaimer.pw experrementummo.pw
63	d25e3a7ed538968e9b78367cd8f8d20f8f…	26148…	/audio/ timerdisclaimer.pw experrementummo.pw
64	7ca44cc3821b27376d9a179cad523d5dc…	20212…	/audio/ ujkiol45.cyou aslopoer45.cyou
65	112ed5790a916786c7ccc38dc5a321a34…	49505…	/audio/ willizoo.website zaxhasshira.uno goodywelli.uno …
66	7bc9ca1d59daf3ba1369bebf24b073c725…	49505…	/audio/ willizoo.website zaxhasshira.uno goodywelli.uno …
67	8c0b7114b76837e81323022ab04faafe29…	49505…	/audio/ willizoo.website zaxhasshira.uno goodywelli.uno …
68	fc19eaeec6edd0d5565f6b3ef1082d36ec…	49505…	/audio/ willizoo.website zaxhasshira.uno goodywelli.uno …

Figure 14. IoCs from IcedID samples.

Conclusion

Thank you for joining us in this overview of malware configurations and why we are working hard to parse this information at scale in Advanced WildFire. Reverse engineering variants of each malware family allow us to build out parsers to extract meaningful and relevant data for all of them at scale.

There is a staggering amount of diversity among payloads in the malware landscape, which makes the task of supporting them all more or less impossible. Where possible, we use metrics-based approaches to prioritize focus on the malware families and variants most relevant to our customers. In this ongoing area of research, our team will continue to expand support for new malware families and variants.

Palo Alto Networks customers receive protections from threats such as those discussed in this post with Advanced WildFire.

Indicators of Compromise

05a3a84096bcdc2a5cf87d07ede96aff7fd5037679f9585fee9a227c0d9cbf51

Additional Resources

Updated May 17, 2023, at 6:00 a.m. PT.