2 - 4 October, 2024 / Dublin, Ireland

BYTEING BACK: DETECTION, DISSECTION AND PROTECTION AGAINST MACOS STEALERS

Patrick Wardle

Objective-See, USA

patrick@objective-see.com

ABSTRACT

Malware is often named and grouped by its end goals: ransomware locks users' files and demands a ransom, while adware often displays invasive ads. Stealers, the topic of today's paper, are no different. These insidious threats, as their name implies, rarely persist, instead stealthily stealing a wide range of sensitive information, such as passwords, browser cookies, and more from infected hosts. Given that such stealers are worryingly prolific and present unique detection challenges, their impact cannot be overstated.

This paper delves into the topic of stealers targeting *macOS*, and aims to provide a comprehensive understanding of several of the common specimens, as well as offering effective heuristic-based methods of proactive protection.

PART I: MACOS STEALERS

In the first part of this paper, we'll explore several recent *macOS* stealers, focusing largely on the most prolific: Atomic. For each, we'll highlight their infection vectors as well as methods of data collection and exfiltration. And what about persistence, a relevant aspect of the analysis of any malicious code? Well, for stealers that do persist, we will of course cover their persistence mechanisms. However, a common and rather unique trait of stealers is their tendency to avoid persistence altogether for two simple reasons. First and foremost, stealers don't need to hang around. Once they've collected and exfiltrated data such as the victim's passwords, cookies, and login information, their job is done! Second, by choosing not to persist, not only is their development simplified, but their stealthiness is significantly increased. Persistence is a rather noisy event that is trivial to detect heuristically, so avoiding it helps stealers remain undetected.

It's worth noting that stealers are, of course, not solely a *macOS* problem, but rather something that affects the entire industry. For example, stealers have been fingered as arguably the root cause of the recent breaches impacting a myriad of companies' data hosted on *Snowflake* [1].

By definition, stealers steal sensitive information. Let's briefly expand upon this, before diving into specific stealers. In a blog post [2] the security researcher Phil Stokes rather comprehensively detailed the most common type of data stolen from *Mac* users by stealers. First and foremost, Phil notes that stealers have an overwhelming propensity for targeting cookies, specifically, those related to sessions. Such cookies may provide attackers access to user accounts, including in some cases, accounts protected by multi-factor authentication.

The system and user keychains (stored in **.keychain** or **.keychain-db** files), are also targeted by the majority of stealers. Armed with the user's password (which, as we'll see, stealers often just ask for), attackers can easily decrypt these keychains in order to access password, keys, certificates, and other secure information.

As noted by Phil, besides targeting cookies, keychains, and the user's password, stealers may seek to capture other browsing data (such as browsing history and browser-saved passwords), SSH keys, and even the contents of the clipboard (pasteboard)!

It is worth noting that, frankly speaking, the current generation of stealers is largely uninteresting from a technical point of view: they leverage well-known infection vectors, standard (if any) obfuscation approaches, and accomplish the goals of accessing sensitive user data in the most prosaic of ways by leveraging standard file I/O APIs. And for security mechanisms such as TCC (that aim to protect sensitive data) that stealers may encounter along the way, most simply take the rather noisy route of simply bothering the user until they acquiesce. Therefore, here, we take a somewhat higher-level approach to avoid getting lost in the rather unimportant technical details, instead pointing out commonalities that will provide a fundamental understanding of these threats. This, of course, conveniently lays the foundation for the second part of this research, which focuses on detection and protection. However, for the interested reader, we will provide citations to detailed technical analysis reports for each sample.

Note: The samples discussed in this paper are available in Objective-See's macOS malware repository [3], allowing for continued analysis and testing against security products.

Atomic (AMOS)

Though adware has traditional been the most common threat facing *macOS* users, in recent years the Atomic stealer has shot to the top of the charts (based on submissions to *VirusTotal*), as shown in Figure 1.

As the most prolific *macOS* stealer, and one that has seen multiple evolutions, its discussion is a logical starting point for this paper.

In April 2023, Duy Phuc Pham posted a short thread on *X* (formerly *Twitter*), both detailing the discovery of Atomic and providing a brief technical overview [4].

The sample masqueraded as a card game, and was distributed via a disk image (.dmg).

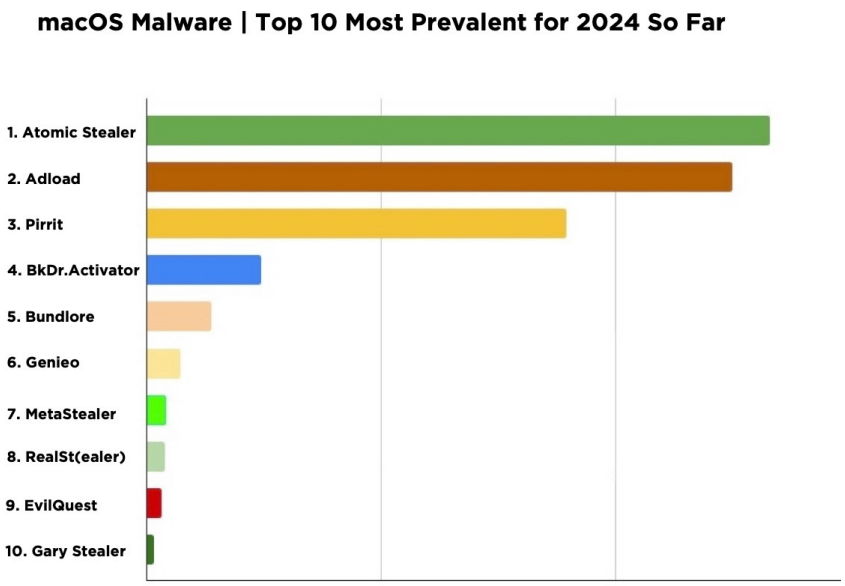


Figure 1: Top macOS threats (image credit: Phil Stokes, via @philofishal).

Infection vector

In terms of its infection vector Phil Stokes notes that, as the Atomic stealer conforms to a malware-as-a-service (MaaS) model, buyers of the malware are responsible for its distribution to end-user machines: 'Payload distribution is left up to the crimeware actor renting the [malware] package, so methods vary, but so far observed samples have been seen masquerading as installers for legitimate applications.' [5]

Another report from *Malwarebytes* [6] notes that the stealer has also been distributed via malvertising, for example displaying malicious ads for social media network *TradingView*. If an unsuspecting user clicks on one of the ads they will be redirected to a website purporting to provide a legitimate version of the *TradingView* application. Instead it's a disk image containing Atomic.

More recently, researchers such as Alden Schmidt have discovered it being distributed by sites that attempt to masquerade as legitimate ones [7]. In Figure 2, for example, we see a malicious site masquerading as *Homebrew*, that instead will serve up a disk image containing the Atomic stealer.

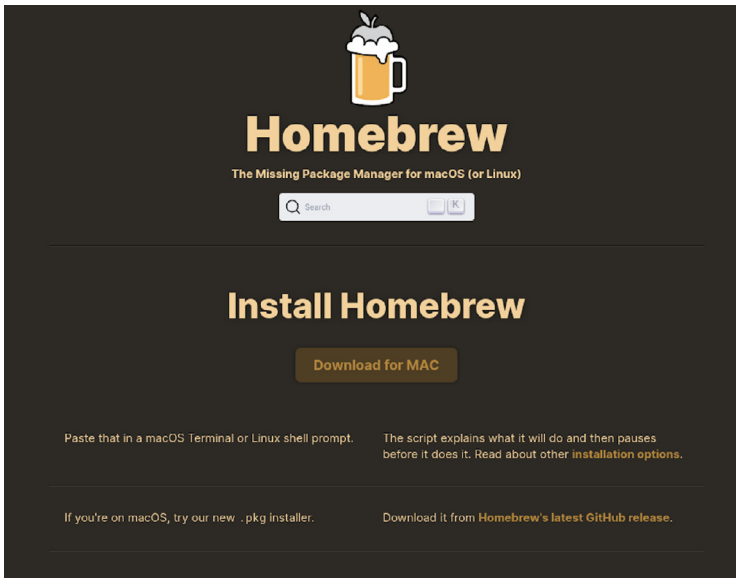


Figure 2: A malicious site serving up the Atomic stealer masquerades as Homebrew (image credit: Alden Schmidt).

In all of these cases, a user will ultimately have to run the malware in order for the infection to commence. And in order to side-step *macOS*'s *Gatekeeper*, once again the malware needs assistance from the user:

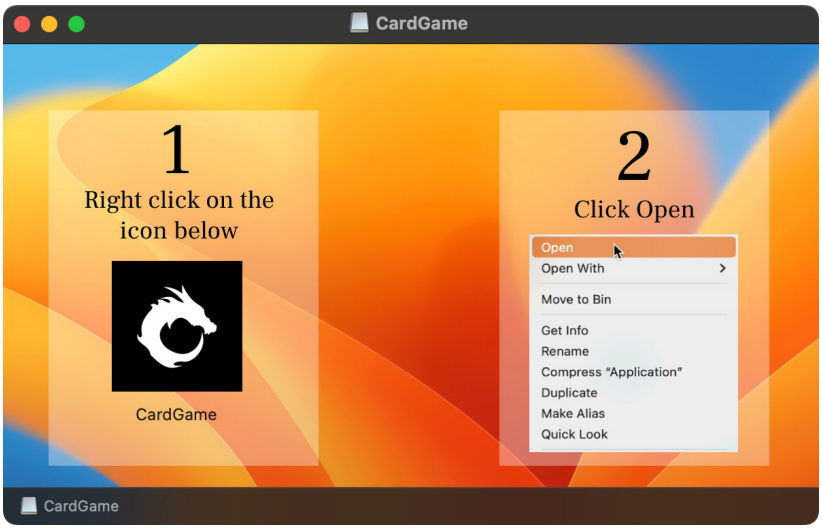


Figure 3: The malware needs assistance from the user in order to side-step Gatekeeper.

As the stealer is only signed with an ad-hoc signature, this step is needed otherwise *Gatekeeper* (which normally only allows signed and notarized items) will block its execution.

```
% codesign -dvvv /Volumes/CardGame/CardGame.app
Executable=/Volumes/CardGame/CardGame.app/Contents/MacOS/My Go Application.app
Identifier=a.out
Format=app bundle with Mach-O universal (x86_64 arm64)
CodeDirectory v=20400 size=52414 flags=0x20002(adhoc,linker-signed)
...
Signature=adhoc
```

Such an infection vector is, yes, both 'high touch', requiring significant user interaction, and quite common in other unsophisticated *Mac* malware families such as adware. Moreover, as we'll see in part II of this paper, it allows us to build effective detection heuristics.

Capabilities

Similar to other stealers, Atomic is designed to collect sensitive data and the exfiltrate it to the attackers' remote servers, as well articulated in the aforementioned *Malwarebytes* report:

> 'The attacker's goal is to simply run their program and steal data from victims and then immediately exfiltrate it back to their own server.'

The main types of information that Atomic targets are browser data (cookies, logins, passwords, stored credit cards), data from popular cryptocurrency extensions, and information stored in the user's keychain.

Though the variant found masquerading as a card game is written in Go, and is thus somewhat difficult to analyse statically, the variant that was distributed via the fake *TradingView* instead appears to be written in C++ and thus is much easier to analyse. Moreover, as this variant doesn't obfuscate its strings, it is easy to gain insight into its capabilities:

```
% strings "/Volumes/Trading View/Trading View"
osascript -e 'display dialog "macOS needs to access System settings
Please enter your password." with title "System Preferences" ...
...
dscl /Local/Default -authonly
You entered invalid password.
system_profiler SPHardwareDataType
VMware
Apple Virtual Machine
/Library/Keychains/login.keychain-db
```

```
/Chromium/Chrome
/Chromium/Chrome/Local State
/Library/Application Support/Firefox/Profiles/
/Gecko/Firefox
cookies.sqlite
/Gecko/Firefox/Cookies
formhistory.sqlite
/Gecko/Firefox/Autofills

Starcoin
CardWallet
TronWallet
CryptoAirdrop

ibnejdfjmmkpcnlpebklmnkoeoihofec
nkbihfbeogaeaoehlefnkodbefgpgknn
bocpokimicclpaiekenaeelehdjllofo

BuildID=
&user=
185.106.93.154
POST /sendlog HTTP/1.1
```

For example, we can see that the malware will display a fake password prompt (via **osascript**) in order to get the user's password, so that it can then access and dump the keychain.

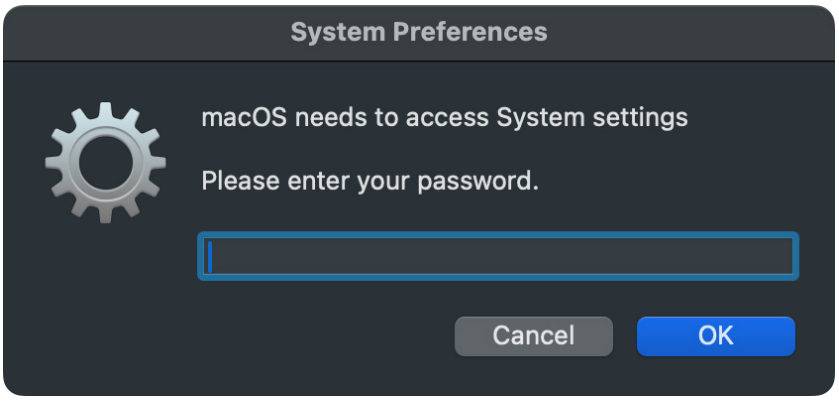
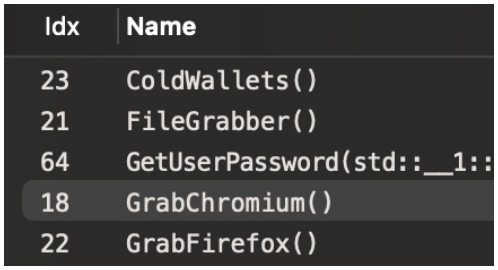


Figure 4: Fake password prompt.

Other embedded strings are related to grabbing data from cryptocurrency wallets (by name, or extension uuid). Finally, we see the embedded address of the attacker's remote server: 185.106.93.154.

The researcher who uncovered Atomic, Duy Phuc Pham, also pointed out on *X* that the malware also 'targets *Chrome*, *Firefox*, *Brave*, *Edge*, *Vivaldi*, *Yandex*, *Opera*, and *OperaGX*. It collects data like autofills, passwords, cookies, and wallets.'

If we load the malware's binary in a disassembler, we can see the names of the functions related to extracting browser data, such as 'GrabChromium' and 'GrabFirefox'.



Idx	Name
23	ColdWallets()
21	FileGrabber()
64	GetUserPassword(std::__1::
18	GrabChromium()
22	GrabFirefox()

Figure 5: Embedded function names in Atomic stealer.

These function names are also viewable via *macOS*'s **nm** utility, though make sure to pipe its output through **c++filt** to demangle their names:

```
% nm "/Volumes/Trading View/Trading View" | c++filt

0000000100004c34 t systeminfo()
0000000100005814 t ColdWallets()
0000000100004de8 t FileGrabber()
0000000100005284 t GrabFirefox()
00000001000033fc t GrabChromium()
00000001000077c4 t GetUserPassword(std::__1::basic_string, std::__1::allocator>)
...
```

The implementation of such functions is rather straightforward. For example, if we take a peek at the decompilation of the GrabChromium function, we can see that, as its name implies, it targets various *Chromium*-based web browsers to collect sensitive data. Strings referenced in the disassembly including 'Cookies', 'Login Data', 'Autofill' and 'Wallets' highlight the information it's after.

Anti-analysis logic

The initial versions of Atomic stealer have limited anti-analysis logic (subsequent versions implement XOR-based string obfuscation) [8]. However, it will attempt to detect if it's running within a virtual machine and if so, exit prematurely.

In order to detect execution within a virtual environment Atomic stealer invokes a function named systeminfo. As the following disassembly shows, this invokes a helper function, exec, to first execute the command **system_profiler** with a single argument: **SPHardwareDataType**.

It then checks if the output from this command contains strings related to virtual machine environments including 'VMware' and 'Apple Virtual Machine':

```
int systeminfo() {
    ...
    exec("system_profiler SPHardwareDataType");
    r0 = std::basic_string::find(..., "VMware", 0x0);
    r8 = r0 + 0x1;

    if ((r8 & 0x1) == 0x0) {
        r0 = std::basic_string::find(..., "Apple Virtual Machine", 0x0);
       r8 = r0 + 0x1;
    }

    if (r8 != 0x0) {
      exit(0x1b);
    }
    ...
```

We can observe the execution of the **system_profiler** utility in a process monitor:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "pid" : 4344
    "name" : "system_profiler",
    "path" : "/usr/sbin/system_profiler",

    "arguments" : [
      "system_profiler",
      "SPHardwareDataType"
    ],
    ...
}
```

In a virtual analysis machine we can see that if we manually execute this command, the following is returned:

```
% system_profiler SPHardwareDataType
Hardware:

    Hardware Overview:

      Model Name: Apple Virtual Machine 1
      Model Identifier: VirtualMac2,1
      Chip: Apple M2 Max (Virtual)
      ...
```

Which means that, due to the presence of the 'Apple Virtual Machine' string, the malware will detect it is running within a virtualized environment and thus exit. This is confirmed in a debugger, where the malware will exit with the value 0x1b:

```
(lldb) "/Volumes/Trading View/Trading View"
...
Trading View`systeminfo:
->  0x100004cb4 <+128>: bl     0x1000180f4     ; symbol stub for: exit

(lldb) reg read $x0
x0 = 0x000000000000001b

(lldb) continue
Process 4957 exited with status = 27 (0x000001b)
```

To bypass this anti-VM logic, one could either patch the malware's binary image, perhaps simply NOP-ing out the call to systeminfo, or in a debugger simply skip its execution (via the **register write $pc <new address>** debugger command).

Atomic continues to evolve, though such evolution appears to be largely driven by detection of the original variants (for example by *Apple*'s *XProtect* signatures). This concludes our analysis of Atomic. You can read more about Atomic in the writeups we have already cited.

Let's now briefly discuss other stealers. While we won't delve as deeply into each one, it's important to note that, conceptually, the core stealing logic of most stealers is largely the same. Despite their differences and lack of direct relation, their approaches to stealing data follow similar patterns.

MetaStealer

Discovered by *SentinelOne*, MetaStealer seems not to target individuals (and their cryptocurrency wallets), but rather businesses and thus business-related data [9].

Infection vector

As is the case with other stealers, MetaStealer spreads via social engineering, requiring a high level of user interaction in order to infect *macOS* systems.

In their analysis report, *SentinelOne* researchers noted that, based on the names of the distributed file, such as 'Advertising terms of reference (MacOS presentation).dmg' and 'OfficialBriefDescription.app.zip', they believed the targets were, in fact, businesses – or at the very least, employees.

Their report also contained reference to an account from one of the targets, which highlights how attackers posed as a client in order to target potential victims:

> 'I was targeted by someone posing as a design client, and didn't realize anything was out of the ordinary. The man I'd been negotiating with on the job this past week sent me a password protected zip file containing this DMG file, which I thought was a bit odd. Against my better judgement I mounted the image to my computer to see its contents. It contained an app that was disguised as a PDF, which I did not open and is when I realized he was a scammer.' -MetaStealer target (via *SentinelOne*).

It's also worth noting that MetaStealer (including the sample we cover here, OfficialBriefDescription.app) is not signed:

```
% codesign -dvv MetaStealer/OfficialBriefDescription.app
MetaStealer/OfficialBriefDescription.app: code object is not signed at all
```

This means that the user would have to jump through various hoops to run the (malicious) application, as *macOS* security mechanisms block unsigned programs by default.

Capabilities

Similar to other stealers, MetaStealer, well, steals information. As it's written in Go, static analysis is somewhat complicated, however as function names are neither stripped nor obfuscated, they give us some insight into the stealer capabilities of the malware, such as its interest in the keychain.

```
Address    Type  Name
0x17259…   P     _miIjdJshZ.(*WzpdHCf).DumpKeyChain
0x17260…   P     _miIjdJshZ.(*WzpdHCf).UploadKeychain
0x17269…   P     _miIjdJshZ.(*WzpdHCf).DecryptKeychain
```

Figure 6: Embedded function names in MetaStealer.

Besides attempting to access the keychain (which would require the victim's password), other functions, such as one named 'GetFiles', provide a mechanism to collect arbitrary files which then may be exfiltrated to the attacker's servers, found at api.osx-mac.com and builder.osx-mac.com.

You can read more about MetaStealer in [9].

JaskaGO

Next we examine JaskaGO. This is a cross-platform stealer targeting both *Windows* and *macOS*. The fact it is cross-platform, coupled with the fact that it persists and supports a wide range of taskable commands, makes it somewhat unique.

JaskaGO was discovered by *AT&T* research labs [10] and, as noted by Phil Stokes, is referred to internally by *Apple* (specifically *macOS*'s *XProtect*) as CherryPie [11].

Infection vector

In their analysis report, *AT&T* researchers stated:

> 'As the malware use of file names resembling well-known applications (such as "Capcut_Installer_Intel_M1.dmg" ...) suggest a common strategy of malware deployment under the guise of legitimate software in pirated application web pages.'

This mimics what we've seen as the infection vectors for other stealers. And again, as the malware is not signed, the user must be coached in order to side-step *macOS* code-signing requirements:

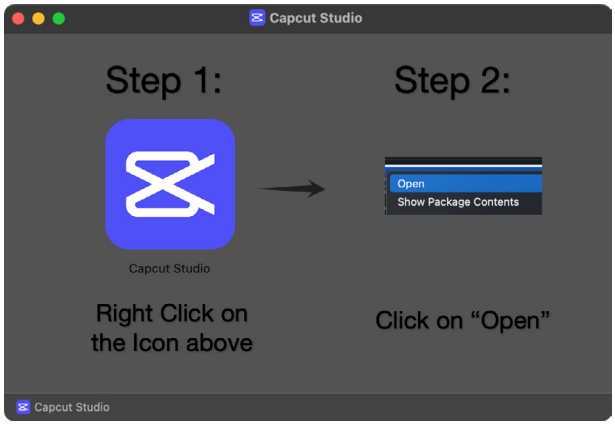


Figure 7: JaskaGO's installation instructions to side-step macOS's Gatekeeper.

Persistence

The vast majority of stealers don't persist, which, as we noted, is one of the reasons they present unique challenges when it comes to detection. However JaskaGO is an exception, as it persists as a launch item. We can see this by disassembling its **service.(*darwinLaunchdService).getServiceFilePath** method (which is called from the aptly named **service.(*darwinLaunchdService).Install** method):

```
service.(*darwinLaunchdService).getServiceFilePath
...
lea     rdi, a20060102150405_0+1EBDh ; "/Library/LaunchAgents/service"
lea     r8, aCookieacceptco+0BAh ; ".plist"
```

```
...
call    runtime_concatstring4

...
lea     rbx, a20060102150405_0+2B19h ; "/Library/LaunchDaemons/Init"
lea     r8, aCookieacceptco+0BAh ; ".plist"
...
call    runtime_concatstring3
```

In the above disassembly, you can see it will persist as either a launch agent (*service.plist*) or launch daemon (*Init.plist*). It chooses the latter if it's running as root; elevated permissions are required to install a launch daemon.

Similar to other stealers, JaskaGO, well, steals information. Reversing the malware reveals its stealing logic is implemented in the 'grinch' class. For example, to extract information stored by the victims' browsers such as cookies and credentials, the methods include:

```
% nm "Capcut Studio.app/Contents/MacOS/Capcut Studio"  | c++filt
...
0000000001457be0 t _motionapp/client/grinch.(*Browser).BrowserExists
0000000001456460 t _motionapp/client/grinch.(*Browser).GetChromiumProfileData
0000000001456fa0 t _motionapp/client/grinch.(*Browser).GetFirefoxProfileData
0000000001457a00 t _motionapp/client/grinch.(*Browser).GetLogins
00000000014575a0 t _motionapp/client/grinch.(*Browser).readDatabaseFile
0000000001457c40 t _motionapp/client/grinch.(*Grincher).GetBrowsersCreds
...
000000000195c6f0 b _motionapp/client/grinch.BrowserList
```

Also, the stealer can (with the user's password) dump the keychain via a method called **mac/system.GetKeychain**.

The *AT&T* research labs report also noted that, besides also grabbing the keychain, the stealer:

> '...searches for browsers crypto wallets extension... In addition, it supports receiving a list of wallets to search for and upload to the server. The malware can [also] receive a list of files and folders to exfiltrate.'

In the same **grinch** class we find the methods that implement this:

```
% nm "Capcut Studio.app/Contents/MacOS/Capcut Studio"  | c++filt
...
00000000014598e0 t _motionapp/client/grinch.(*Grincher).GetWallets
0000000001459a80 t _motionapp/client/grinch.(*Grincher).getWalletData
0000000001459d00 t _motionapp/client/grinch.(*Grincher).processWallet
...
0000000001458180 t _motionapp/client/grinch.(*FileGripper).GetFiles
0000000001458620 t _motionapp/client/grinch.(*FileGripper).filterFilesByExtension
0000000001458840 t _motionapp/client/grinch.(*FileGripper).filterFilesBySize
```

Finally, JaskaGO also has the ability to run arbitrary commands, via methods such as **system.RunCommandWithSudo** and **system.RunExecutableWithSudo**, or download new binaries (payloads via **_motionapp/client/modules.downloadBinary**).

Let's look at the implementation of the **system.RunCommandWithSudo** method:

```
lea     r8, aTruemapEeppTxt+10h ; "sudo
...
lea     r8, aITvrruueeaalls+30h ; "-S
call    os_exec_Command
call    os_exec__ptr_Cmd_Start
call    fmt_Fprintf
...
lea     rax, aBreakOutsideRa+204h ; "failed to write password"
```

At its core, the **system.RunCommandWithSudo** method invokes the **os_exec_Command** and **os_exec_Cmd_Start** methods to execute a command via sudo. Strings referenced early in the function show it first builds the string: **sudo -S**. To add the password, it uses the **fmt_Fprintf** methods. Of course, this means the malware already has the user's password (which it also needs to dump the keychain, and more).

Taking into account JaskaGO's persistence and additional taskable capabilities, such as the download and execution of (additional) binaries, it is safe to say it's one of the more fully-featured, and thus dangerous *macOS* stealers.

And all the others

There are many other stealers targeting *macOS* including Cuckoo, MacStealer, GoSorry, PureLand and Realst, which we'll mention briefly and, where relevant, cite reports containing additional analysis of each specimen.

Cuckoo

Cuckoo was discovered and subsequently analysed by researchers Adam Kohler and Christopher Lopez at *Kandji* [12].

Of note is that it persists itself as launch agent via a property list named 'com.dumpmedia.spotifymusicconverter.plist' that (re)runs the stealer every 60 seconds:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" …>
<plist version="1.0">
<dict>
   <key>Label</key>
   <string>com.user.loginscript</string>
   <key>ProgramArguments</key>
   <array>
      <string>/Users/User/…/DumpMediaSpotifyMusicConverter</string>
   </array>
   <key>StartInterval</key>
   <integer>60</integer>
</dict>
</plist>
```

Cuckoo is interested in the usual types of sensitive data, but also enumerates installed applications, collects files matching common file extensions (.txt, .doc, .pdf, .jpg, etc), and contains screen capture logic.

MacStealer

This stealer was discovered by researchers at *Uptycs*, who uncovered the malware for sale on the dark web [13]. Written in Python, though compiled as a Mach-O executable for distribution, MacStealer is fairly standard in terms of what it collects. From the names of the .pyc files we can see it is interested in browser cookies and passwords, keychains, data from various cryptocurrency wallets, and file matching common file extensions.

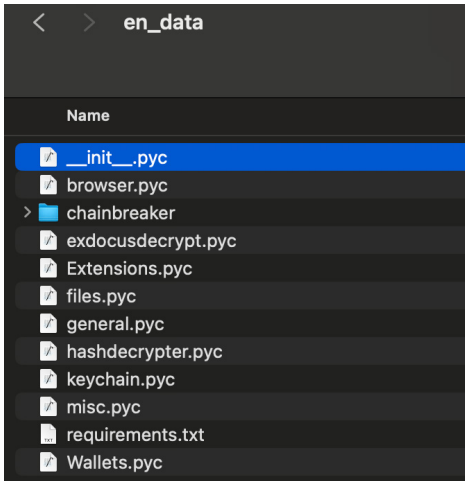


Figure 8: MacStealer's (.pyc) files give insight into its capabilities.

It's worth repeating that the apparent logic in the files' names are, of course, also confirmed by continued analysis.

In addition to exfiltrating the data to http://mac.cracked23.site, MacStealer also uploads it to the attacker's personal *Telegram* bot, according to *Uptycs* researchers.

GoSorry

GoSorry, first mentioned on *X* by @malwrhunterteam, is a dynamic library named 'liboptimizer.dylib' [14].

Similar to other stealers, GoSorry is designed to steal a variety of sensitive information from its victims, though it seems to focus solely on browser data and (browser-based) cryptocurrency wallets.

As it's written in Go, it contains various rather verbose method names, which reveal the capabilities of the malware. These are prefixed with **__sorryforthat**:

```
% nm GoSorry/liboptimizer.dylib | c++filt
...
0000000000371fc0 t _sorryforthat/internal/browingdata/bookmark.(*ChromiumBookmark).Parse
00000000003732e0 t _sorryforthat/internal/browingdata/bookmark.(*FirefoxBookmark).Parse

00000000009474d0 d _sorryforthat/internal/browingdata/common.CHROMIUM_WALLETS
00000000009474f0 d _sorryforthat/internal/browingdata/common.EDGE_WALLETS
0000000000947510 d _sorryforthat/internal/browingdata/common.FIREFOX_WALLETS
0000000000947530 d _sorryforthat/internal/browingdata/common.OPERA_WALLETS

0000000000376560 t _sorryforthat/internal/browingdata/cookie.(*ChromiumCookie).Parse
00000000003770a0 t _sorryforthat/internal/browingdata/cookie.(*FirefoxCookie).Parse

0000000000377a00 t _sorryforthat/internal/browingdata/creditcard.(*ChromiumCreditCard).Parse
0000000000378200 t _sorryforthat/internal/browingdata/creditcard.(*YandexCreditCard).Parse

0000000000378da0 t _sorryforthat/internal/browingdata/download.(*ChromiumDownload).Parse
0000000000379580 t _sorryforthat/internal/browingdata/download.(*FirefoxDownload).Parse

000000000037b9e0 t _sorryforthat/internal/browingdata/history.(*ChromiumHistory).Parse
000000000037c0e0 t _sorryforthat/internal/browingdata/history.(*FirefoxHistory).Parse

00000000003cd540 t _sorryforthat/internal/browingdata/password.(*ChromiumPassword).Parse
00000000003cf0c0 t _sorryforthat/internal/browingdata/password.(*FirefoxPassword).Parse

0000000000955680 b _sorryforthat/internal/browser.braveProfilePath
00000000009556a0 b _sorryforthat/internal/browser.chromeProfilePath
00000000009556d0 b _sorryforthat/internal/browser.edgeProfilePath
00000000009556e0 b _sorryforthat/internal/browser.firefoxProfilePath
0000000000955730 b _sorryforthat/internal/browser.torProfilePath
0000000000955750 b _sorryforthat/internal/browser.yandexProfilePath
```

PureLand and Realst

Uncovered and analysed by @Iamdeadlyz, these two seemingly related stealers predominantly focus on stealing information that would give attackers access to users' cryptocurrency wallets [15].

For example, disassembling the PureLand stealer binary reveals 'search'-related functions that, based on their names, appear to be interested in finding common cryptocurrency wallets to plunder, as shown in Figure 9.

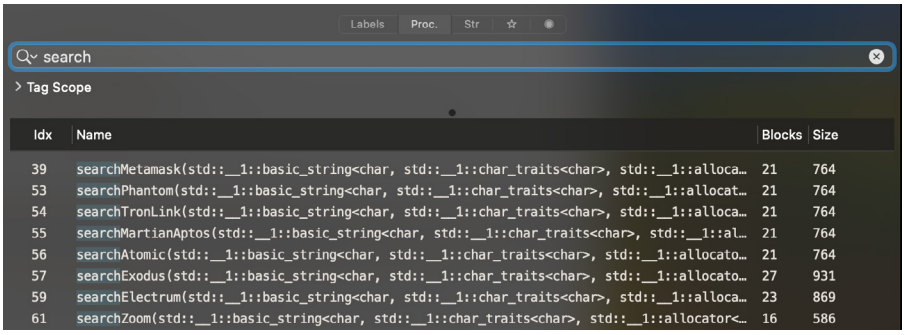


Figure 9: PureLand's search-related methods.

Realst largely leans on publicly available scripts to perform its stealing, which includes accessing the contents in the *macOS* keychain as well as decrypting (and stealing) cookies from browsers, as noted by Iamdeadlyz:

> 'The game.py script's original filename is firefox_decrypt.py by unode - https://github.com/unode/firefox_decrypt. Firefox Decrypt is a tool to extract passwords from profiles of Mozilla (Fire/Water)fox™, Thunderbird®, SeaMonkey® and derivatives.
>
> 'The installer.py script is a combination of scripts from n0fate's chainbreaker - https://github.com/n0fate/chainbreaker Chainbreaker can be used to extract the information [passwords, certs, etc.] from an OSX keychain.'

PART II PROTECTION (AND DETECTION)

The list and impact of stealers targeting *macOS* is ever growing, affecting both end-users and enterprises. Such stealers are widely available either for sale on the 'dark web', or offered via MaaS. Others are directly created and distributed by prolific cybercriminals groups. Some are written in Python, while others are written in Rust, Go, or Swift. Most do not persist. The breadth and ephemerality of stealers poses significant challenges to traditional signature-based detection approaches.

However, if we take a step back and make the rather obvious observation that all stealers steal, and if we can craft a heuristic around this commonality, we should be able to detect all stealers in one fell swoop.

First though, let's focus on generically blocking their execution in the first place.

On recent versions of *macOS*, software must be notarized in order to run, unless the user explicitly exempts it. However, to be notarized, the developer must submit the binary to *Apple*, which scans it for malware. If none is found, *Apple* approves it via notarization. This, of course, poses an expected conundrum to malware authors: submit the malicious binaries to *Apple* in the hope that they will inadvertently be notarized, or rely on users to exempt the malware? Though *Apple* has inadvertently notarized malware on occasion, this is exceedingly rare. Moreover, even in this case, once *Apple* realizes its error, it can quickly revoke either the notarization ticket or the developer ID altogether, blocking the malware and any other binaries signed with the same code-signing certificates. Moreover, from the notarization submission, *Apple* will (already) have the malicious binary in its possession, allowing it to be fully analysed, and from this analysis *Apple* can improve its notarization approval process or even globally release *XProtect* detection signatures. With the risk-reward calculation clearly skewed, by design, toward *Apple*, most malware authors, including those creating stealers, opt for the user-exception approach.

While covering the infection vectors of *macOS* stealers we discussed this approach, illustrating how the stealer would instruct the user simply to control/right click and then select 'Open' to side-step *macOS* notarization requirements. Even the most recent versions of the prolific Atomic stealer (still) take this approach, setting the background of its distribution medium, a disk image, to an image containing instructions to the user:

Figure 10: In order to side-step macOS's security mechanisms, stealers require user interaction.

Though *macOS* will still warn the user that the application is not notarized and thus should not be run, there is an option to allow it. And if the user selects this option, the malware will unfortunately run and steal all the things.

As the vast majority of legitimate software is notarized, a very powerful yet effective protection heuristic is simply to block the execution of any item that is not notarized, regardless of the user's wishes. Though, yes, slightly draconian, this will stop all current stealers in their tracks!

To block unnotarized code we can leverage *Apple*'s *Endpoint Security*. The framework allows clients to subscribe to events and, in many cases, examine the event to determine if it should be allowed to commence. Let's dive directly into code that allows us to authorize process executions, and then show how to examine such processes to ascertain their notarization status, and finally block any that are non-notarized.

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_AUTH_EXEC};

es_new_client_result_t result =
es_new_client(&client, ^(es_client_t *client, const es_message_t *message) {
        es_process_t* process = nil;
        es_string_token_t* procPath = nil;


        process = message->event.exec.target;
        procPath = &process->executable->path;


        printf("\nevent: ES_EVENT_TYPE_AUTH_EXEC\n");
        printf("process: %.*s\n", (int)procPath->length, procPath->data);

        es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
});

if(ES_NEW_CLIENT_RESULT_SUCCESS != result) {
        //TODO: handle error/exit
}

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

In the above code we first create an array containing the *Endpoint Security* events we're interested in subscribing to. In this simplified example, we specify only one event: **ES_EVENT_TYPE_AUTH_EXEC**. This event will be delivered to us prior to a new process being allowed to execute, giving our code an opportunity to examine it and make a decision about its execution. More on this shortly.

Next we use the es_new_client API to create a new *Endpoint Security* client. This client is required for all subsequent calls into *Endpoint Security*. The API takes as its last parameter a block that will automatically be invoked by the framework whenever any of the events we're interested in occur. For now, as we're going to subscribe to **ES_EVENT_TYPE_AUTH_EXEC** events, we extract the process that's about to be spawned and print out its path. Then, as we're dealing with an **AUTH_*** event, we must tell the framework whether to allow or deny (block) the process, via the es_respond_auth_result API. For now, we always invoke it with **ES_AUTH_RESULT_ALLOW** to allow all processes. Shortly we'll show how to block any process that isn't notarized.

Note that the es_new_client can fail for a variety of reasons, including if the code isn't running with root (**ES_NEW_CLIENT_RESULT_ERR_NOT_PRIVILEGED**), does not have full disk access (**ES_NEW_CLIENT_RESULT_ERR_NOT_PERMITTED**), or isn't adequately entitled (**ES_NEW_CLIENT_RESULT_ERR_NOT_ENTITLED**).

Finally, we subscribe to the *Endpoint Security* event of interest via the es_subscribe API, which as you can see takes our client, the events, and their count. After this function has returned, **ES_EVENT_TYPE_AUTH_EXEC** events will begin streaming into the handler block we passed to es_new_client.

At this point, we can see each new process that is spawned, for example on my developer box processes related to git:

```
...
event: ES_EVENT_TYPE_AUTH_EXEC
process: /Applications/Xcode.app/Contents/Developer/usr/bin/git

event: ES_EVENT_TYPE_AUTH_EXEC
process: /Applications/Xcode.app/Contents/Developer/usr/libexec/git-core/git-remote-http
```

However, we're just authorizing all processes, so now let's show how to determine if a process is notarized, and if not, how to block it.

```
SecRequirementRef isNotarized = nil;

SecRequirementCreateWithString(CFSTR("notarized"), kSecCSDefaultFlags, &isNotarized);

if(errSecSuccess == SecCodeCheckValidity(dynamicCode, kSecCSDefaultFlags, isNotarized)) {
        //process is notarized
}
```

It's straightforward to check if a process is notarized using *Apple*'s code-signing APIs. As shown above, first create a 'requirement reference' that is initialized with the string 'notarized'. This is then passed to the **SecCodeCheckValidity** API, which will return a zero (**errSecSuccess**), if and only if the item is conformant to the specified requirement, which here, is whether the item is notarized or not.

Note: Not shown is the creation of the dynamic code reference, which can be created by passing the process' audit token to the **SecCodeCopyGuestWithAttributes** API. And, yes, this code reference should also be validated wholly via a call to **SecCodeCheckValidity** before any subsequent checks are taken.

If the item is not notarized, we can simply invoke the **es_respond_auth_result ES_AUTH_RESULT_DENY** to tell *macOS* to deny the processes execution:

```
SecRequirementRef isNotarized = nil;

SecRequirementCreateWithString(CFSTR("notarized"), kSecCSDefaultFlags, &isNotarized);

if(errSecSuccess == SecCodeCheckValidity(dynamicCode, kSecCSDefaultFlags, isNotarized)) {
        //process is notarized, so allow
        es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
}
else {
       //process is *not* notarized, so block
       es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
}
```

It should be noted that, as is often the case, in reality things are a bit more complicated. For example, core *macOS* binaries (named 'platform binaries' in *Apple* parlance) are not notarized, but of course should be allowed. Applications from *Apple*'s official *Mac App Store* also are not notarized, but generally you'll want to allow them, as they undergo the same, if not more stringent, review process as notarized items. One approach that at least takes into account, or rather ignores platform binaries, is only checking items that have been downloaded from the internet, which carry the **com.apple.quaratine** extended attribute. This approach is taken by my *BlockBlock* tool, which has a mode to block non-notarized items. As it is open-source, you can consult its source code [16] to see a full implementation, including how to use the undocumented APIs in the **libquarantine** dynamic library to determine if an item has the quarantine attribute set.

If we add this code to a proof of concept 'blocker', and then attempt to execute an instance of the Atomic stealer, it is resoundingly blocked:

```
...
event: ES_EVENT_TYPE_AUTH_EXEC
process: /Volumes/CardGame/CardGame.app/Contents/MacOS/CardGame

process 'CardGame' has the quarantine attribute set
process 'CardGame' is not notarized, so will block!

returning ES_AUTH_RESULT_DENY
```

With the ability to block any non-notarized process from executing, we can, as noted, stop all existing stealers in their tracks as currently none are notarized. However, *Apple* is not infallible and inadvertently notarizes malware from time to time. As such, blocking non-notarized items should not be seen as a panacea. Thus, let's end this paper by showing how to leverage the mute-inversion capabilities of *Endpoint Security* to write some code that protects the data that stealers are after, by only authorizing access to trusted programs.

One of the oft overlooked capabilities of *Endpoint Security* is mute inversion. In a nutshell, this allows one to monitor specific programs, files/directories. Here, we'll walk through some example code that monitors the user's entire Documents directory, that could easily be extended to only allow access by trusted or recognized processes.

Note: Why is this referred to as mute inversion? Well, the original version of Endpoint Security did not provide such a capability. Instead, there was only the ability to mute or ignore events from specified processes and or files/directories. It was realized, shortly thereafter, that the inverse of this would be supremely useful as well. However, instead of providing a whole new set of APIs, Apple wisely realized that a single API, ***es_invert_muting****, could then toggle the muting logic within the well understood muting APIs.*

Much of the code we'll show here is similar to the code in the previous example where, after *Endpoint Security* events were specified and an *Endpoint Security* client was created, the code subscribed to the events (via a call to **es_subscribe**).

```
es_client_t* client = NULL;

es_event_type_t events[] = {ES_EVENT_TYPE_AUTH_OPEN};

NSString* consoleUser = (__bridge_transfer NSString *)SCDynamicStoreCopyConsoleUser(NULL,
NULL, NULL);

NSString* docsDirectory = [NSHomeDirectoryForUser(consoleUser)
stringByAppendingPathComponent:@"Documents"];

es_new_client(&client, ^(es_client_t *client, const es_message_t *message) {

        es_string_token_t* procPath = nil;
        es_string_token_t* filePath = nil;

        procPath = &message->process->executable->path;

        filePath = &message->event.open.file->path;
        printf("\nevent: ES_EVENT_TYPE_AUTH_OPEN\n");
        printf("process: %.*s\n", (int)procPath->length, procPath->data);
        printf("file path: %.*s\n", (int)filePath->length, filePath->data);

        //TODO: add code to check process, only approve authorized ones
        // for now we authorize all, but use ES_AUTH_RESULT_DENY to block

        es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);

});

es_unmute_all_target_paths(client);
es_invert_muting(client, ES_MUTE_INVERSION_TYPE_TARGET_PATH);
es_mute_path(client, docsDirectory.UTF8String, ES_MUTE_PATH_TYPE_TARGET_PREFIX);

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

After specifying the *Endpoint Security* events of interest – here just the **ES_EVENT_TYPE_AUTH_OPEN** event – we look up the console user's name via **SCDynamicStoreCopyConsoleUser** so that we can then build a path to their Documents directory. Such an approach is necessary as programs that utilize *Endpoint Security* must run as root, thus convenience APIs such as **NSHomeDirectory** will return the root user's director, which is not what we actually want.

The callback block passed to the **es_new_client** API simply prints out the name of the process attempting to open a file in the directory we're watching, as well as the actual file. In this example we simply return **ES_AUTH_RESULT_ALLOW** via the **es_respond_auth_result** API, which will authorize all accesses. Of course, you'll want to examine the process attempting to access the file, and if it's not trusted, block it by instead returning **ES_AUTH_RESULT_DENY**.

Next up is the actual code related to the mute inversion. First, and very importantly, we invoke **es_unmute_all_target_paths**. This call is needed as, by default, *Apple* mutes certain system processes which we want to remain muted after we invert muting. We then tell *Endpoint Security* that here we want to mute everything but the path (we're) about to specify. As such, we invoke the **es_invert_muting** API. To specify the directory to protect we then invoke the **es_mute_path** API (which, as muting is inverted, will actually mute events for everything else). We pass in **ES_MUTE_PATH_TYPE_TARGET_PREFIX** as we want to watch files whose path matches a specified prefix (here, the console user's Documents directory).

Finally, we call **es_subscribe** to subscribe to events of interest, kicking off the protection.

Though not shown here, if we add additional 'is the process trusted' logic, we now have a powerful protection mechanism capable of generically thwarting stealers, including Atomic stealer:

```
protecting directory: /Users/User

event: ES_EVENT_TYPE_AUTH_OPEN
```

```
process: /Volumes/CardGame/CardGame.app/Contents/MacOS/CardGame
file path: /Users/User/Documents
...
process 'CardGame' is not trusted, so will block!
returning ES_AUTH_RESULT_DENY
```

CONCLUSION

After a brief introduction to stealers, the first half of this research paper described several of the most prolific and widespread stealers currently targeting *macOS* users. This analysis illustrated commonalities among different samples, laying the foundation for building powerful detection heuristics. In the second half of the paper, we explored these detection approaches, specifically highlighting examples that utilized *Apple*'s *Endpoint Security* framework to block non-notarized processes and protect the user's Documents directory. With these robust protections in place, our hope is that we can now decisively turn the tide against these nefarious and highly impactful threats.

REFERENCES

[1] https://x.com/JohnHultquist/status/1800153633792889145.

[2] Stokes, P. Session Cookies, Keychains, SSH Keys and More | 7 Kinds of Data Malware Steals from macOS Users. SentinelOne. 22 March 2023. https://www.sentinelone.com/blog/session-cookies-keychains-ssh-keys-and-more-7-kinds-of-data-malware-steals-from-macos-users/.

[3] Objective-See. macOS Malware Repository. https://github.com/Objective-see/Malware.

[4] https://x.com/phd_phuc/status/1651001139750420480.

[5] Stokes, P. Atomic Stealer | Threat Actor Spawns Second Variant of macOS Malware Sold on Telegram. SentinelOne. 3 May 2023. https://www.sentinelone.com/blog/atomic-stealer-threat-actor-spawns-second-variant-of-macos-malware-sold-on-telegram/.

[6] Segura, J. Mac users targeted in new malvertising campaign delivering Atomic Stealer. Malwarebytes. 6 September 2023. https://www.malwarebytes.com/blog/threat-intelligence/2023/09/atomic-macos-stealer-delivered-via-malvertising.

[7] Schmidt, A. An Infostealer's Brewin': Cuckoo & AtomicStealer Get Creative. 14 May 2024. https://alden.io/posts/infostealers-a-brewin/.

[8] DD. Breaking down Atomic MacOS Stealer (AMOS). 5 March 2024. https://medium.com/@dineshdevadoss04/breaking-down-atomic-macos-stealer-amos-8cd5eea56024.

[9] Stokes, P. macOS MetaStealer | New Family of Obfuscated Go Infostealers Spread in Targeted Attacks. SentinelOne. 11 January 2024. https://www.sentinelone.com/blog/macos-metastealer-new-family-of-obfuscated-go-infostealers-spread-in-targeted-attacks/.

[10] Caspi, O. Behind the scenes: JaskaGO's coordinated strike on macOS and Windows. LevelBlue. 18 December 2023. https://cybersecurity.att.com/blogs/labs-research/behind-the-scenes-jaskagos-coordinated-strike-on-macos-and-windows.

[11] https://x.com/philofishal/status/1737351740586872840.

[12] Kohler, A.; Lopez, C. Malware: Cuckoo Behaves Like Cross Between Infostealer and Spyware. Kandji. 30 April 2024. https://blog.kandji.io/malware-cuckoo-infostealer-spyware.

[13] Trivedi, S. MacStealer: Unveiling a Newly Identified MacOS-based Stealer Malware. Uptycs. 24 March 2023. https://www.uptycs.com/blog/macstealer-command-and-control-c2-malware.

[14] https://x.com/malwrhunterteam/status/1651160948780986368.

[15] Iamdeadlyz. PureLand – A Fake Project Related to the Sandbox Malspam. 7 March 2023. https://iamdeadlyz.medium.com/pureland-a-fake-project-related-to-the-sandbox-malspam-13b9abe751d1.

[16] BlockBlock. https://github.com/objective-see/BlockBlock.