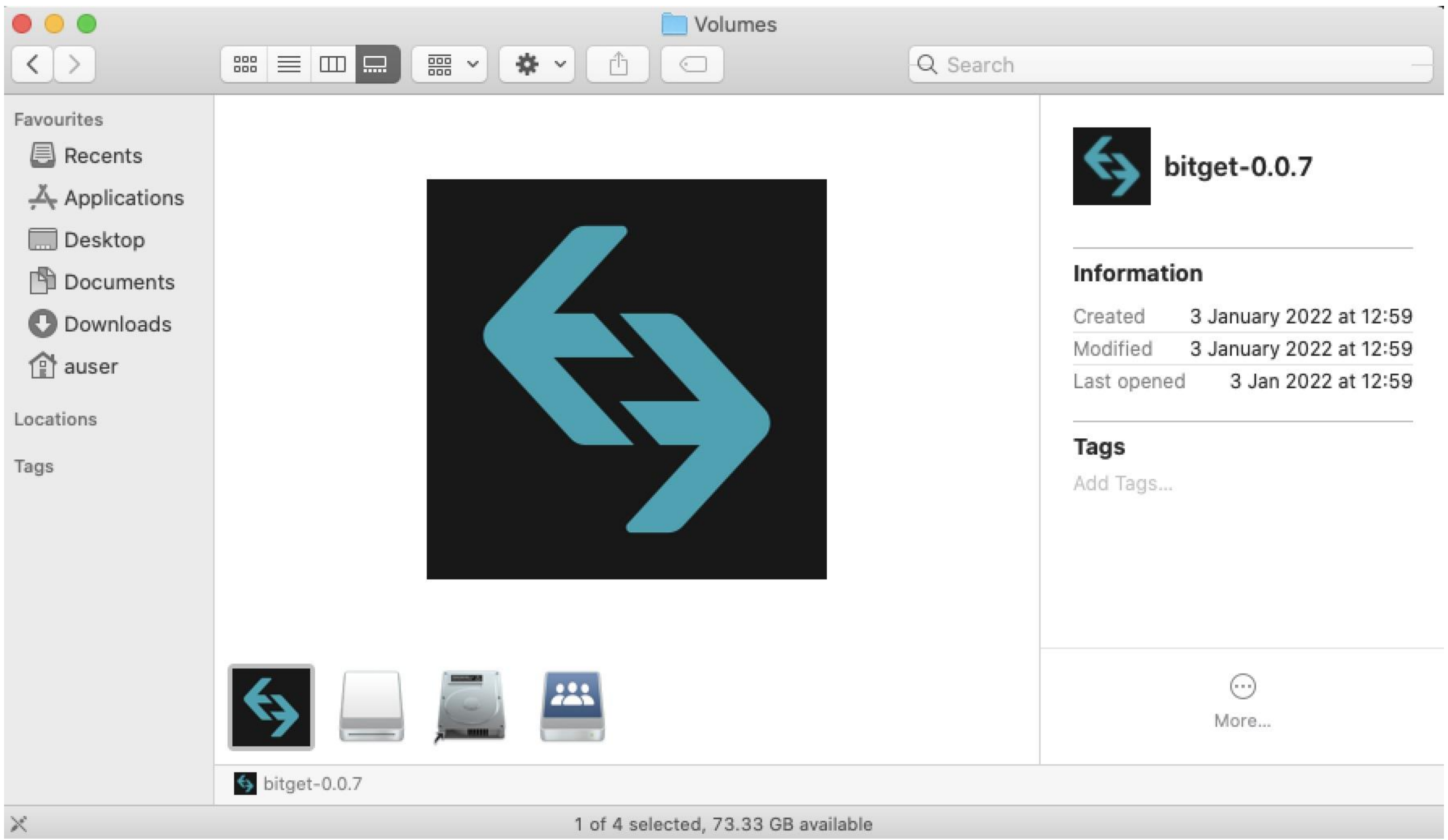
By Dinesh Devadoss and Phil Stokes

Researchers looking into a new APT group targeting gambling sites with a variety of cross-platform malware recently identified a version of oRAT malware targeting macOS users and written in Go. While neither RATs nor Go malware are uncommon on any platform, including the Mac, the development of such a tool by a previously unknown APT is an interesting turn, signifying the increasing need for threat actors to address the rising occurrence of Macs among their intended targets and victims. In this post, we dig deeper into the technical details of this novel RAT to understand better how it works and how security teams can detect it in their environments.

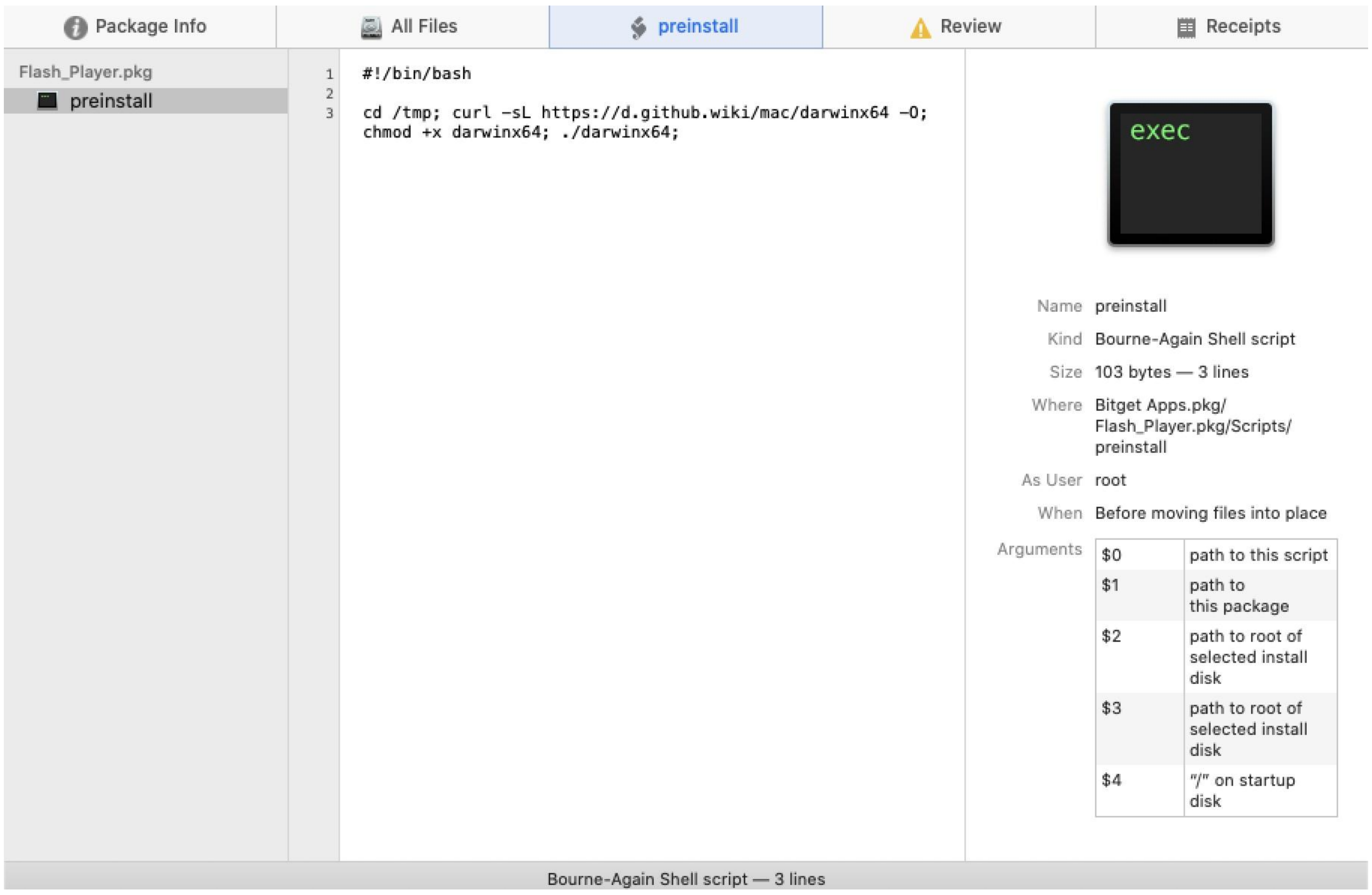
oRAT Distribution

The oRAT malware is distributed via a Disk Image masquerading as a collection of Bitget Apps. The disk image contains a package with the name Bitget Apps.pkg and the distribution identifier `com.adobe.pkg.Bitget`.



The disk image and installer package are notable for two reasons: neither has a valid developer signature, and the latter doesn't actually install any files and only contains a preinstall script.

The preinstall script is a succinct bash shell script whose purpose is to deliver a payload to the `/tmp` directory, give the payload executable permissions, and then launch it.



Precisely what kind of lure the threat actors use to convince targets to download and launch the dropper is unknown at this time, but given that the target would need to override default security warnings from Gatekeeper, it is likely either that the users are sourcing the malware from an environment where this is typical (e.g., a 3rd-party software distribution site that regularly delivers unsigned software) or users have been pre-groomed to bypass Gatekeeper during a social engineering engagement of some kind.

In either case, the fact that there's no deliverable from the user's perspective is a risky gamble on the part of the threat actors. After running the installer and finding that it did not provide whatever they were expecting, users are likely to become suspicious. This might suggest the campaign was broadly targeted and that the threat actors were playing a numbers game, happy to sweep up opportunistic infections as they occurred.

The oRAT Payload

Things get more interesting when we examine the `darwinx64` payload dropped in the `/tmp` folder. The binary doesn't define any Symbols, and outputting the list of Sections tells us that the file has been packed with UPX.

```
auser@reversing-lab-10 orats % rabin2 -s darwinx64
[Symbols]

nth paddr vaddr bind type size lib name
―――――――――――――――――――――――――――――――――――――――
auser@reversing-lab-10 orats % rabin2 -S darwinx64
[Sections]

nth paddr          size vaddr          vsize perm name
―――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x000003b0  0x3afc50 0x01a123b0  0x3afc50 -r-x 0.__TEXT.upxTEXT
```

Packed files like this are opaque to static analysis, but fortunately standard UPX is very easy to unpack thanks to the UPX utility itself. Dumping the strings tells us that it was packed with UPX 3.96, the most recently released version available.

The packed binary is around 3MB in size, but after unpacking we are presented with a massive ~10MB file. Such large file sizes are typical of cross-platform malware, particularly when binaries are compiled in Go, since they contain the entire run-time for the language along with a number of supporting libraries.

Fortunately, from a reverse engineering perspective, we can easily ignore most of the standard code that is common to all Go bins and focus on what is unique to the sample at hand. For IDA Pro users, see here; for r2 users, we can start by printing out a list of the functions flagged with `sym._main`.

```
[0x01465f80]> afl~sym._main
0x014ae260   18 426   -> 425   sym._main.Daemon
0x014ae420   10 273            sym._main.createDaemon
0x014ae540    7 341            sym._main.isRunning
0x014ae6a0    7 212            sym._main.createPidFile
0x014ae780    3 267            sym._main.watchSignal
0x014ae8a0   28 923            sym._main.main
0x014aec40    1 9              sym._main.watchSignal.func1
0x014aec60   19 414            sym._main.main.func1
```

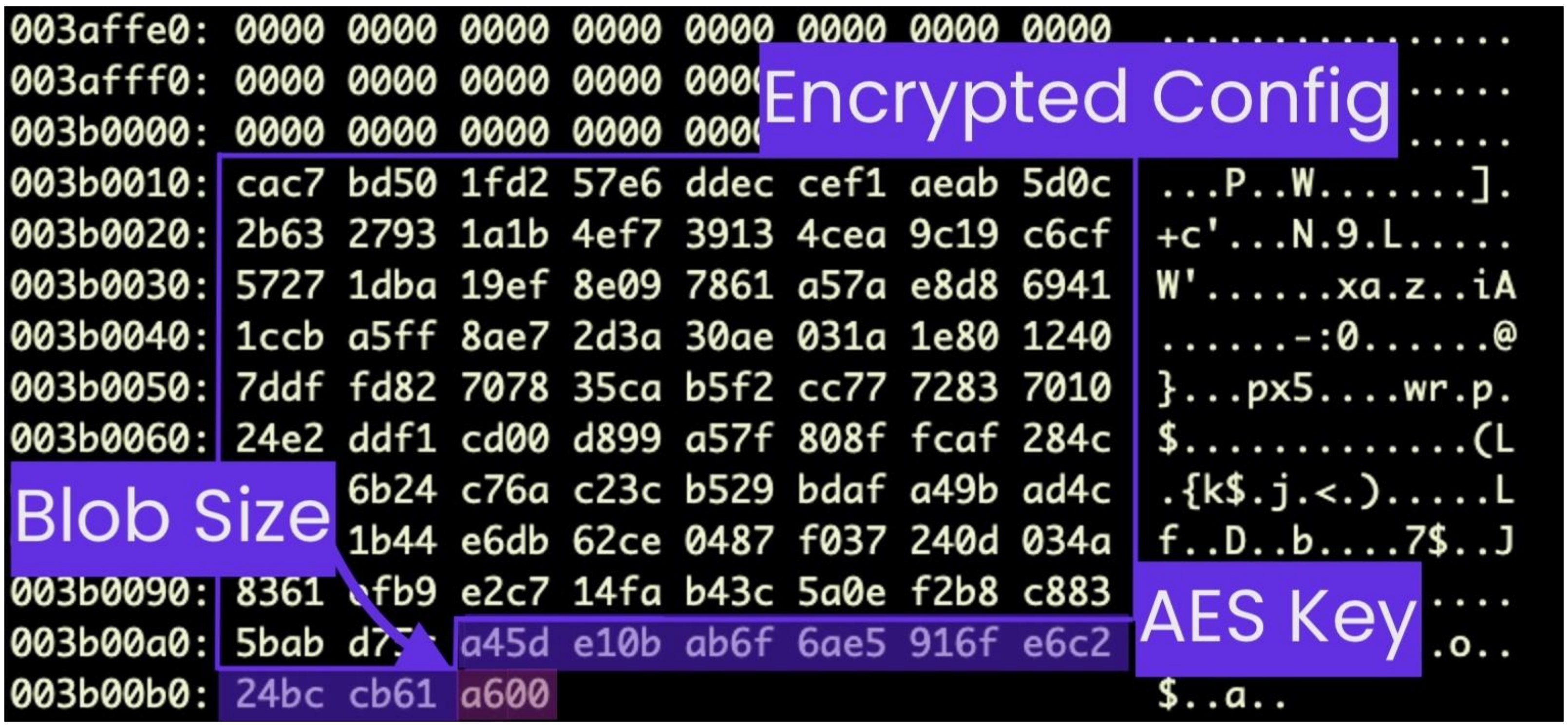
In Go binaries, the program code entrypoint is at main.main, and we can work our way through there to see what other functions, packages and modules are called. Below, we see that the main.main function calls out to another custom package, `orat_utils`.

```
[0x0106b840]> s sym._main.main
[0x014ae8a0]> pds
0x014ae8cf "RK_DEBUGReceivedSETTINGSSHA1-RSASHA3-224SHA3-256SHA3-384SHA3-512SSH-2.0-SameSiteSaturdayT
agbanwaTai_ThamTai_VietThursdayTifinag"
0x014ae8e3 call sym._os.Getenv
0x014ae8f4 call sym._orat_utils.LoadConfig
0x014ae94f call sym._log.Println
0x014ae960 call sym._runtime.newobject
0x014ae972 "stcpstepsubesudpsup1sup2sup3supesynctag:tcp6truetypeuArruarruintunixuseruumlvaryvoidwaitw
ithwrapxn--yumlzetazwnj ...\n H_T= H_a="
0x014ae985 "127.0.0.1:28888400 Bad Request476837158203125: cannot parse <invalid Value>ASCII_Hex_Digi
tAccept-EncodingAccept-LanguageClientA"
0x014ae995 int64_t arg_50h
0x014ae99e int64_t arg_48h
0x014ae9b0 int64_t arg_40h
0x014ae9b9 int64_t arg_38h
0x014ae9b9 "RK_NETReady.RejangSCHED STREETScaronServerStart.StringSundaySyriacTMPDIRTai_LeTangutTarge
tTeluguThaanaThreadTypeMXTypeNSUacuteU"
0x014ae9cd call sym._os.Getenv
0x014ae9f2 int64_t arg_48h
0x014ae9f2 "RK_ADDRRadicalRefererSHA-224SHA-256SHA-384SHA-512SharadaShavianSiddhamSignal SinhalaSogdi
anSoyomboSubjectSwapperTERM=%sTagalogT"
0x014aea06 call sym._os.Getenv
[0x014ae8a0]> _
```

The `orat_utils` package contains several interesting functions and gives us an entry into understanding how the RAT works.

```
[0x014ae8a0]> afl~orat_utils!stkobj
[0x014ae8a0]> afl~orat_utils | grep -v stkobj
0x01465aa0   14 517  -> 515  sym._orat_utils.Unpack
0x01465cc0   13 698  -> 693  sym._orat_utils.Decrypt
0x01465f80   23 572  -> 567  sym._orat_utils.LoadConfig
0x014661c0   14 682          sym._orat_utils.GetRandomString
0x01466480    5 210          sym._orat_utils.GenerateSigner
0x01466560    9 517          sym._orat_utils.Pipe
0x01466780   11 823  -> 802  sym._orat_utils.Forward
0x01466ac0    7 172          sym._orat_utils.FileExists
0x01466b80    3 89           sym._orat_utils.Pipe.func1.1
0x01466be0    2 377          sym._orat_utils.Pipe.func1
0x01466d80    3 261          sym._orat_utils.Pipe.func2
0x01466ea0    3 261          sym._orat_utils.Pipe.func3
0x01466fc0   11 458  -> 456  sym._orat_utils.Forward.func1
0x014671a0   11 180          sym._type..eq.orat_utils.NetConfig
0x01467260   24 374          sym._type..eq.orat_utils.Config
```

Of particular interest is the `LoadConfig` function. This is used to parse a blob of data appended to the binary which turns out to be an encrypted malware configuration. The encrypted data at the end of the unpacked binary occupies 166 bytes and consists of the data, an AES key, and two bytes representing the entire blob size.



Once decrypted, the blob turns out to contain configuration data for the malware C2.

```
{
  "Local": {
    "Network": "sudp",
    "Address": ":5555"
  },
  "C2": {
    "Network": "stcp",
    "Address": "darwin.github.wiki:53"
  },
  "Gateway": false
}
```

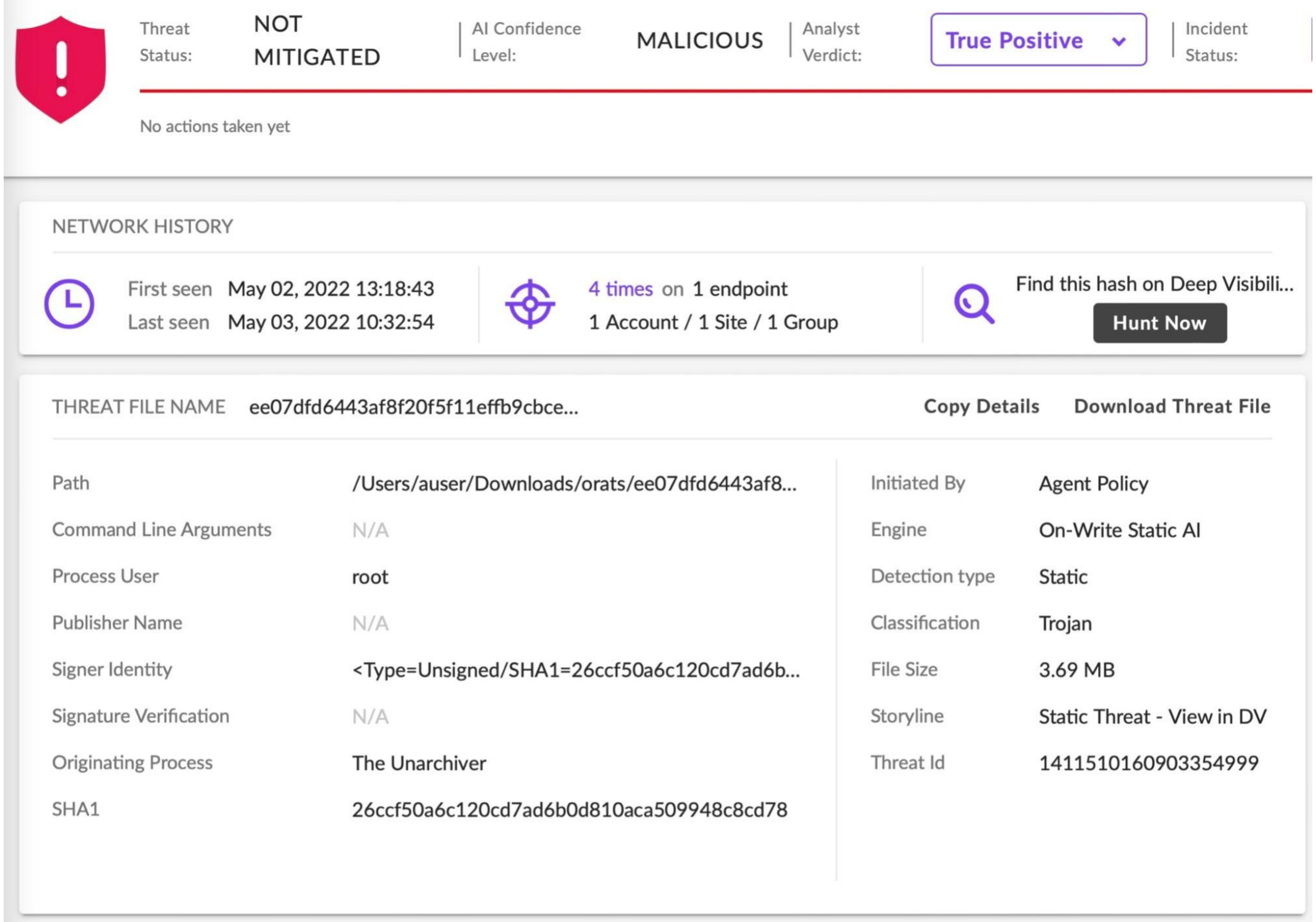
After the malware decodes the config, it calls into `sym._orat_cmd_agent.app` and begins a number of loops through `sys._orat_protocal.Dial`. Depending on the config, it will call one of `orat_protocol.DialTCP`, `orat_protocol.DialSTCP` or `orat_protocol.DialSUDP` to establish a connection. The TCP protocols leverage [smux](#) while the SUDP protocol leverages [QUIC](#). The malware loops with a sleep cycle of 5 seconds as it waits for a response.

The `sym._orat_cmd_agent.app` contains the primary RAT functionality of the malware and defines the following functions.

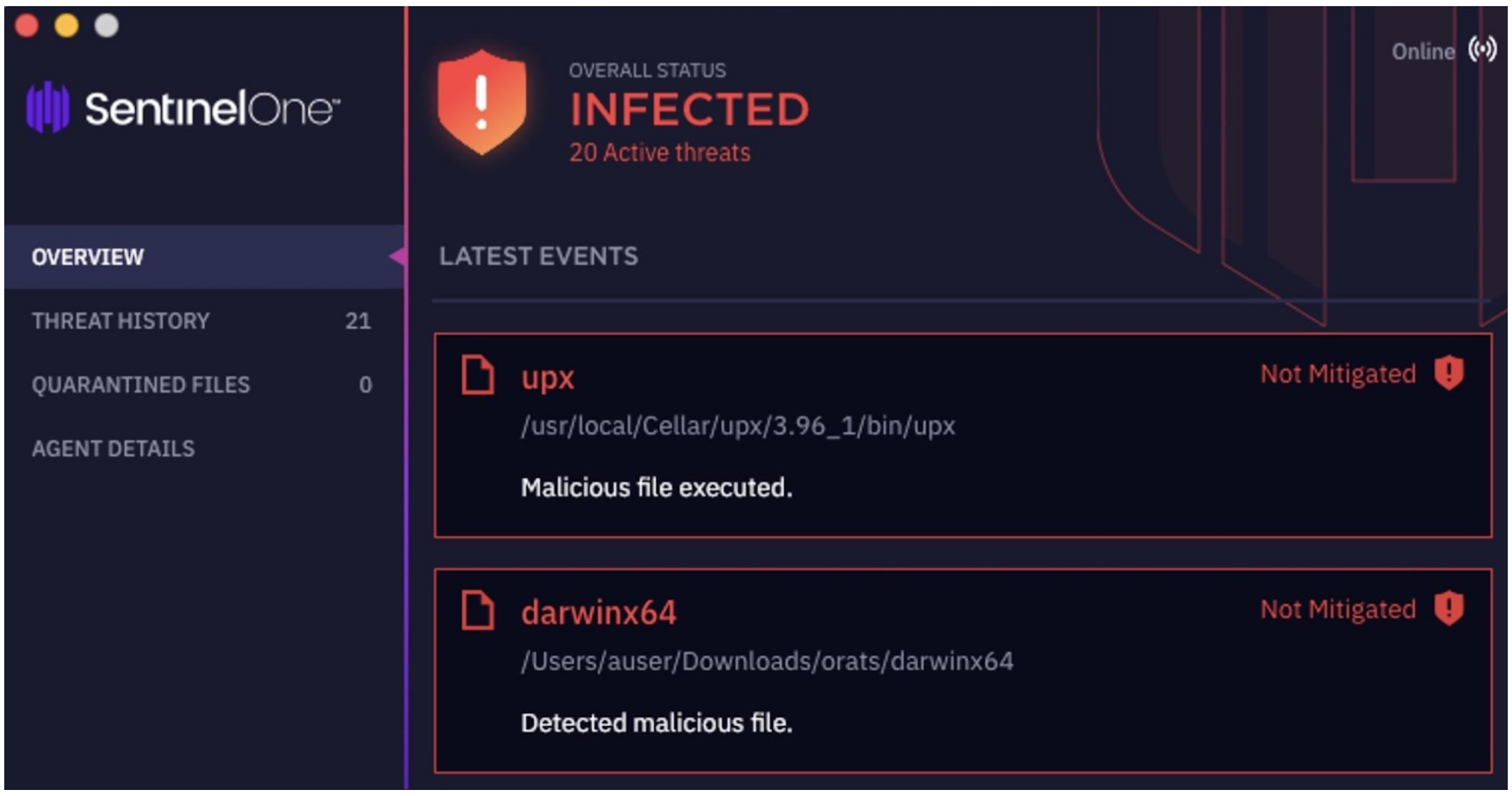
orat/cmd/agent/app.(*App).DownloadFile orat/cmd/agent/app.(*App).Info orat/cmd/agent/app.(*App).Join orat/cmd/agent/app.(*App).KillSelf orat/cmd/agent/app.(*App).NewNetConn orat/cmd/agent/app.(*App).NewProxyConn orat/cmd/agent/app.(*App).NewShellConn orat/cmd/agent/app.(*App).Ping orat/cmd/agent/app.(*App).PortScan orat/cmd/agent/app.(*App).registerRouters orat/cmd/agent/app.(*App).run orat/cmd/agent/app.(*App).Screenshot orat/cmd/agent/app.(*App).Serve orat/cmd/agent/app.(*App).Unzip orat/cmd/agent/app.(*App).UploadFile orat/cmd/agent/app.(*App).Zip

Detecting oRAT in the Enterprise

The SentinelOne agent detects the oRAT payload as malicious when it is written to disk, protecting SentinelOne customers from this threat.



The SentinelOne agent also detects the malware on execution.



For those not protected by the SentinelOne platform, security teams are advised to hunt for artifacts as listed in the Indicators of Compromise section at the end of this post.

Conclusion

The oRAT malware targets macOS users using a combination of custom-written code and public Golang repos. The developers are clearly familiar with using sophisticated features of Go for networking and communications, but due to the simplistic way the malware dropper was packaged, unsigned and with no observable install to distract the victim, it would seem they are less experienced with the challenges of infecting Mac users. Unfortunately, other threat actors have provided plenty of examples from which this new player can learn, and security teams should expect to see any future campaigns from this actor using more sophisticated droppers.

Indicators of Compromise

Filename	SHA1
bitget-0.0.7 (1).dmg	3f08dfafbf04a062e6231344f18a60d95e8bd010

Bitget Apps.pkg	9779aac8867c4c5ff5ce7b40180d939572a4ff55
preinstall	911895ed27ee290bea47bca3e208f1b302e98648
`darwinx64` (packed)	26ccf50a6c120cd7ad6b0d810aca509948c8cd78
`darwinx64` (unpacked)	9b4717505d8d165b0b12c6e2b9cc4f58ee8095a6

Paths

/tmp/`darwinx64`