Objective-See's Blog

objective-see.org/blog/blog_0x78.html

Analyzing DPRK's SpectralBlur

The first malware of 2024 is (already) here!

by: Patrick Wardle / January 4, 2024

📝 👾 Want to play along?
As 'Sharing is Caring' I've uploaded a sample of the malware SpectralBlur.zip to our public macOS malware collection. The password is: infect3d

...please though, don't infect yourself! 🙈

Background

Not three days into 2024 Greg Lesnewich tweeted the following:

> #100DaysofYARA day 03 - talking SpectralBlur, a MacOS (and other OS 🤫) backdoor linked to TA444/Bluenoroff, that I suspect is a cousin of the KandyKorn family our pals at Elastic found! https://t.co/P2TGw98UR6 pic.twitter.com/Y8U3hsjNiF
>
> — Greg Lesnewich (@greglesnewich) January 3, 2024

In both his twitter (err, X) thread and in a subsequent posting he provided a comprehensive background and triage of the malware dubbed `SpectralBlur`. In terms of its capabilities he noted:

> SpectralBlur is a moderately capable backdoor, that can upload/download files, run a shell, update its configuration, delete files, hibernate or sleep, based on commands issued from the C2. -Greg

He also pointed out similarities to/overlaps with the DPRK malware known as `KandyKorn` (that we covered in our "Mac Malware of 2024" report), while also pointing out there was differences, leading him to conclude:

> We can see some similarities ... to the KandyKorn. But these feel like families developed by different folks with the same sort of requirements. -Greg

Greg's writeup, though focusing more on the discovery and triage of the SpectralBlur, is excellent a read, and provides essential background and context for our more indepth analysis.

Have a read:
100DaysofYARA - SpectralBlur
Greg ended his write up, nothing that he hadn't had time yet to fully analyze the sample, but "*if anyone in the community is keen on it, go for it!!*". As I'm rather obsessed with (new) Mac malware I decided to dig in more.

Triage

The sample of SpectralBlur mentioned in his tweets and triage has a SHA-1 hash of `06c8c84fb0a85bdf3520608b0a5c910b77e3b8c1`. Popping over to VirusTotal to grab the sample, we can see that currently it is not flagged as malicious by any of the AV engines:

SpectralBlur on VirusTotal (Scan Date: Aug. 2023)

Triggering a rescan of the malware on VirusTotal shows it is (now) detected by at least one AV engine (ESET).

Moreover its worth noting that the previous scan of the malware (refereneced in the screenshot above) was performed in August 2023. Thus, though at that time no AV engine flagged the item as malicous, between now and then, detections could have been added.

Looking at telemetric data we can see its name is either `.macshare` or `mac.jpg`, and at least on one macOS system it was found in `/Users/Shared/.macshare`.

Prefixing the name with a '.' will hide the file from Finder (unless the viewing of hidden files has been enabled).

This SpectralBlur sample was initially submitted to VirusTotal on `2023-08-16` from Colombia (CO). Interestingly in VirusTotal's telemetric data, we can also see that at least one of Objective-See's tools (which, integrate with VirusTotal, for example to allow users to submit unrecognized files) encountered the malware in the wild too …how cool!

Using `WhatsYourSign` we can see the binary is not signed:

WhatsYourSign shows the malware is not signed
You can also extract codesigning information (or lack thereof) from the terminal via macOS's `codesign` utility:

```
% codesign -dvv SpectralBlur/.macshare
SpectralBlur/.macshare: code object is not signed at all
```

Via the terminal we can also see its a 64bit Intel (`x86_64`) binary:

```
% file SpectralBlur/.macshare
SpectralBlur/.macshare: Mach-O 64-bit executable x86_64
```

Next let’s pull out any embedded strings, as this can give a good idea of the malware’s capabilities and also guide continued analysis:

```
% strings SpectralBlur/.macshare

There is NO WARRANTY, to the extent permitted by law.
%s.d
/dev/null
SHELL
/bin/sh
...
```

Other than some strings related to the shell, not a lot! Maybe others are obfuscated? (We also didn’t show function names or API imports in the `strings` output, as these show up better via `nm`).

Using `nm` we can extract symbols which will include the Malware’s function names, as well as APIs that the malware calls into (“imports”). Let’s start with just function names, which will be found the `__TEXT` segment/section: `__text`. We can use `nm`’s `-s` to limit its output to just a specified segment/section:

```
% nm -s __TEXT __text SpectralBlur/.macshare
0000000100001540 T _hangout
00000001000034f0 T _init
0000000100001570 T _init_fcontext
0000000100001870 T _load_config
0000000100003650 T _main
0000000100002a10 T _mainprocess
0000000100003370 T _mainthread
00000001000031c0 T _openchannel
00000001000029a0 T _proc_die
0000000100001b10 T _proc_dir
0000000100002420 T _proc_download
0000000100002290 T _proc_download_content
00000001000027e0 T _proc_getcfg
00000001000028c0 T _proc_hibernate
00000001000019f0 T _proc_none
00000001000029d0 T _proc_restart
00000001000025a0 T _proc_rmfile
0000000100002860 T _proc_setcfg
0000000100001a90 T _proc_shell
0000000100002930 T _proc_sleep
0000000100001a20 T _proc_stop
00000001000026b0 T _proc_testconn
0000000100002160 T _proc_upload
0000000100002040 T _proc_upload_content
00000001000015f0 T _read_packet
0000000100001930 T _save_config
0000000100001500 T _sigchild
00000001000011f0 T _socket_close
0000000100000d10 T _socket_connect
0000000100001140 T _socket_recv
00000001000010c0 T _socket_send
0000000100000be0 T _wait_read
0000000100001730 T _write_packet
00000001000017e0 T _write_packet_value
0000000100001270 T _xcrypt
```

Looks like functions dealing with a config (e.g., `load_config`), network communications (e.g., `socket_recv`, `socket_send`), and encryption (`xcrypt`). But also then, standard backdoor capabilities implemented (as noted by Greg), in function prefixed with `proc`.

And what about the APIs the malware imports to call into? Again we can use `nm`, this time the `-u` flag:

```
% nm -m SpectralBlur/.macshare
_connect
_dup2
_execve
...
_fork
_fread
_fwrite
...
_gethostbyname
_getlogin
_getpwuid
_getsockopt
_grantpt
...
_ioctl
_kill
...
_pthread_create
_rand
_recv
_send
_socket
_unlink
_waitpid
_write
...
```

From these imports we can surmise that the malware performs file I/O (`fread`, `fwrite`, `unlink`), network I/O (`socket`, `recv`, `send`), and spawning/managing processes (`execve`, `fork`, `kill`).

We'll see these APIs are invoked by the malware often in response to commands. For example, the malware's `proc_rmfile` function invokes the `unlink` API to remove a file:

```
int proc_rmfile(int arg0, int arg1) {
    var_10 = arg1;
    var_18 = var_10 + 0x10;
    ...
    unlink(var_18);
    ...
```

Based on this triage (and Greg's report), we've developed what is likely a reasonable understanding of the malware. Still, its always good to validate triages with continued static, but also dynamical analysis. So …onward!

Analysis

At the start of the malware disassembly it calls into a function named `init`. Here, it builds a path to its config, and then opens it. The path is built by appending `.d` to the malware's binary full path:

```
init(...){

   _sprintf_chk(config, 0x0, 0x41a, "%s.d", malwaresPath);
   ...

   loadConfig(...)
}
```

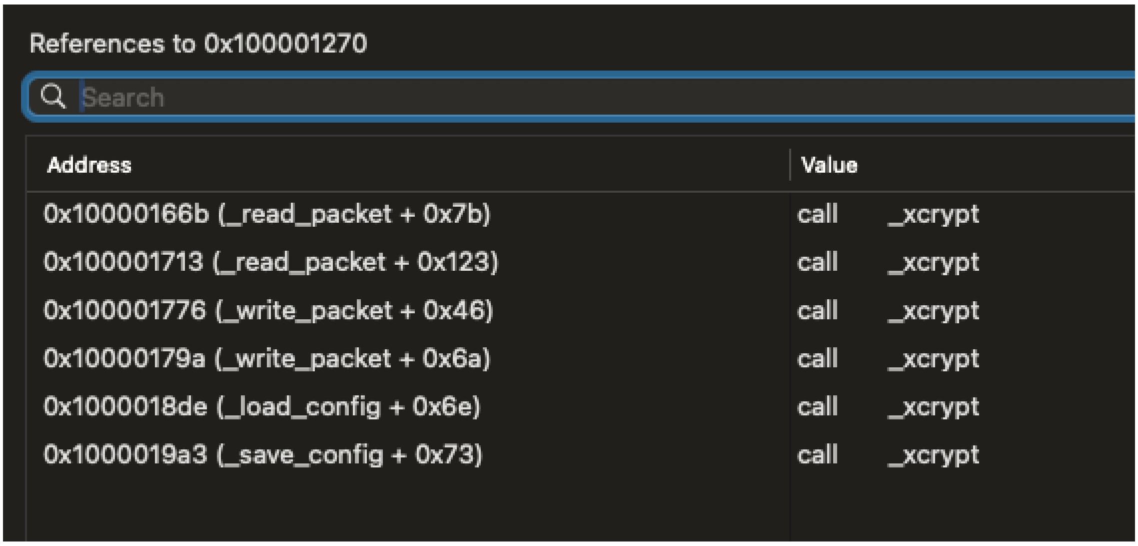
We can confirm this in a debugger, where at a call to `fopen` (in the `load_config` function) the malware will attempt to open the file `macshare.d`, in the directory where the malware is currently running (e.g. `/Users/user/Downloads/`).

```
% lldb /Users/user/Downloads/macshare

(lldb)

* thread #1, queue = 'com.apple.main-thread'
macshare`load_config:
->  0x100001890 <+32>: callq  0x100003d8c               ; symbol stub for: fopen

Target 0: (macshare) stopped.
(lldb) x/s $rdi
0x100008820: "/Users/user/Downloads/macshare.d"
```

We can also see this in a File Monitor:

```
# ./FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter macshare
{
  "event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
  "file" : {
    "destination" : "/Users/user/Downloads/macshare.d",
    "process" : {
        "pid" : 6818,
        "name" : "macshare",
        "path" : "/Users/user/Downloads/macshare"
    }
  }
}
```

By looking at its cross-references (xrefs), we can see the `xcrypt` function is invoked to encrypt/decrypt the malware's config and network traffic:



References to 0x100001270

Address	Value	
0x10000166b (_read_packet + 0x7b)	call	_xcrypt
0x100001713 (_read_packet + 0x123)	call	_xcrypt
0x100001776 (_write_packet + 0x46)	call	_xcrypt
0x10000179a (_write_packet + 0x6a)	call	_xcrypt
0x1000018de (_load_config + 0x6e)	call	_xcrypt
0x1000019a3 (_save_config + 0x73)	call	_xcrypt

XRefs to the xcrypt function

…the `xcrypt` function according to ChatGPT appears to be a custom stream cipher. While static analysis shows that the key may be stored at the start of this config (address `0x100008c3a`), and set to random 64bit value:

```
*qword_100008c3a = sign_extend_64(rand()) + time(0x0) + sign_extend_64(rand() *
rand());
```

Back to the config file, unfortunately, I (currently) don't have access an example config. Thus some of our continued analysis is based solely on static analysis.

Once the `init` function returns (which loaded the config), that malware performs a myriad of actions that appear to complicate dynamic analysis and perhaps detection. This including `fork`ing itself, but also setting up a pseduo-terminal via `posix_openpt` (as noted by Phil Stokes):

> Uses grantpt to set up a pseudo-terminal. Not seen that before. Interesting!
>
> — Phil Stokes ⊏🐠⊐ (@philofishal) January 4, 2024

…this is followed by more forks, execs, and more. Again, if I had to guess, this simply to complicate analysis (and/or perhaps, making it a detached/"isolated" process complicated detections)? We'll also see that the psuedo-terminal is used to execute shell commands from the attacker's remote C&C server.

Regardless we can skip over this all, and simply continue execution (or static analysis) where a new thread (named `_mainthread`) is spawned. After invoking functions such as `openchannel` and `socket_connect` to likely connect to its C&C server (whose address likely would be found in the malware's config: `macshare.d`), it invokes a function named `mainprocess`.

The `mainprocess` function (eventually) invokes the `read_packet` function which appears to return the index of a command. The code in `mainprocess` function then iterates over an array named `_procs` in order to find the handler for the specified command (that I've named `commandHandler` in the below disassembly). The command handler is then directly invoked:

```
int mainprocess(int arg0, int arg1) {

    var_558 = read_packet(...);
    if (var_558 != 0x0) goto loc_100002dfc;

loc_100002dfc:
    var_560 = *(var_558 + 0x8);
    commandHandler = 0x0;
    addrOfProcs = _procs;
    do {
            var_5C1 = 0x0;
            if (*addrOfProcs != 0x0) {
                var_5C1 = (var_568 != 0x0 ? 0x1 : 0x0) ^ 0xff;
            }
            if ((var_5C1 & 0x1) == 0x0) {
                break;
            }
            if (*addrOfProcs == var_560) {
                    commandHandler = *(addrOfProcs + 0x4);
            }
            addrOfProcs = addrOfProcs + 0xc;
    } while (true);


   var_538 = (commandHandler)(var_530, var_558);

}
```

After creating a custom structure (`procStruct`) for this array, we can see each command number and its handler:

```
procs:

0x0000000100008000          struct procStruct {
                                0x1,
                                _proc_none
                            }
0x000000010000800c          struct procStruct {
                                0x2,
                                _proc_shell
                            }
0x0000000100008018          struct procStruct {
                                0x3,
                                _proc_dir
                            }
0x0000000100008024          struct procStruct {
                                0x4,
                                _proc_upload
                            }
0x0000000100008030          struct procStruct {
                                0x5,
                                _proc_upload_content
                            }
0x000000010000803c          struct procStruct {
                                0x6,
                                _proc_download
                            }
0x0000000100008048          struct procStruct {
                                0x7,
                                _proc_rmfile
                            }
0x0000000100008054          struct procStruct {
                                0x8,
                                _proc_testconn
                            }
0x0000000100008060          struct procStruct {
                                0x9,
                                _proc_getcfg
                            }
0x000000010000806c          struct procStruct {
                                0xa,
                                _proc_setcfg
                            }
0x0000000100008078          struct procStruct {
                                0xb,
                                _proc_hibernate
                            }
0x0000000100008084          struct procStruct {
                                0xc,
                                _proc_sleep
                            }
0x0000000100008090          struct procStruct {
                                0xd,
                                _proc_die
                            }
0x000000010000809c          struct procStruct {
```

```
                                     0xe,
                                     _proc_stop
                                 }
0x00000001000080a8               struct procStruct {
                                     0xf,
                                     _proc_restart
                                 }
```

Recall we saw the names of each command handler (`_proc_*`) in the output of `nm`. And, though we can guess the likely capability of eahc command from its name, let's look a few to confirm.

The `proc_rmfile` will remove a file by invoking the `unlink` API. However, we can also see that it first opens the file (`fopen`) and overwrites its contents with zero:

```
int proc_rmfile(int arg0, int arg1) {
    var_4 = arg0;
    var_10 = arg1;
    var_18 = var_10 + 0x10;
    file = fopen(var_18, "rb+");
    if (file != 0x0) {
            fseek(file, 0x0, 0x2);
            var_28 = ftell(file);
            fseek(file, 0x0, 0x0);
            var_30 = 0x5000;
            if (var_28 < var_30) {
                var_30 = var_28;
            }
            var_38 = malloc(var_30);
            _memset_chk(var_38, 0x0, var_30, 0xffffffffffffffff);
            fwrite(var_38, 0x1, var_30, file);
            free(var_38);
            fclose(file);
    }
    rdx = unlink(var_18);
    rax = 0x0;
    if (rdx == 0x0) {
            rax = 0x1;
    }
    return _write_packet_value(var_4, *var_10, rax);
}
```

…each command will also report a result by invoking the malware `write_packet_value` API.

The `proc_restart` will terminate the child process:

```
int main(...)

   call       fork
   mov        dword [childPID], eax

int proc_restart(int arg0, int arg1)

   kill(*childPID, 0x9);
   return write_packet_value(arg0, *arg1, 0x0);
```

Finally, let's look at the `proc_shell`, which executed a command by `write`'ing to the pseudo-terminal that was opened (via `posix_openpt`) previously:

```
int main(...)

   call       posix_openpt
   mov        dword [pt], eax

int proc_shell(...) {
   var_8 = arg0;
   var_10 = arg1;
   if (write(*pt, var_10 + 0x10, strlen(var_10 + 0x10)) <= 0x0) {
           var_4 = _write_packet_value(var_8, *var_10, 0x0);
   }
```

The other commands execute actions consistent with their respective names.

Conclusion

Today we dug into SpectralBlur, a DPRK backdoor. Building upon Greg's research we further detailed its capabilities.

Interested in learning more Mac Malware Analysis Techniques?

—