



Cracking Cisco's Sourcefire licensing system

Jose Gonzalez Krause
email: josef@hackercat.ninja
twitter: [@bitsniper](https://twitter.com/bitsniper)

February 20, 2018

1 The problem

Cisco's Sourcefire system is the IDS/IPS solution offered by this company after the acquisition of Sourcefire, including its network anomaly detection engine, Snort. This IPS solution is one of the more powerful systems available on the market.

The system is composed mainly by two appliances:

- The **sensor** –FirePOWER–, is the IPS itself with Snort, the RNA –Real Network Awareness– engine, *nmap*, the signature database and all the stuff that makes sense on an IPS. This appliance is mainly physical but Cisco offers also a virtual appliance option available on the customer support portal.
- The **manager** –FireSIGHT Management center (FSM)–, is the central administration console, one FSM can have attached multiple sensors, and all the configuration is done here, so as policy creation, firewall rules, object setup, rule edition, etc. Once configured or modified some policy the whole config/rule/stuff package is deployed to the paired sensors. This element can be run as a virtual appliance available on the Cisco customer support portal.

The main problem of Cisco's Sourcefire system is that the hardware is completely useless without a valid license. After buying a sensor on Ebay or scavenging one from a death project or whatever, a license is still needed to make them to work, and yes, these licenses are not exactly cheap...

In this writeup I'll use the following laboratory setup:

- Virtual FSM on version 5.4.1.9
- Physical sensor 3D2000 on version 5.4.0.9
- Physical sensor 3D7110 on version 5.4.0.9

In this writeup I'll make use of some custom tools and scripts, all this code is available on my [git](https://dev.hackercat.ninja/hcninja/sflicense) —<https://dev.hackercat.ninja/hcninja/sflicense>— server.

At least, but not less important, the bypass techniques exposed in this paper are also applicable to the latest versions of Sourcefire sensors and FSMs —Tested on FSM version 6—. According to Cisco, neither its ASA nor the new Firepower Threat Defense appliances are susceptible to the demonstrated license bypass. However, I am not able to confirm or deny this as I haven't had the chance to test those systems.

Lets go!

2 The license file

The first step was to find out what a license file actually contains, fortunately I'm currently working with multiple FireSIGHT appliances and I have access to plenty of original valid licenses, let's look at one:

```
Device: [REDACTED]
Features: PROTECT+CONTROL

--- BEGIN SourceFire Product License :

bW [REDACTED] Y0
Ok [REDACTED] NL
cm [REDACTED] Q3
MT [REDACTED] tT
tV [REDACTED] bb
0G [REDACTED] DQ
Mi [REDACTED] Gs
QD [REDACTED] va
MV [REDACTED] Xi
Ga [REDACTED] Zq
ei [REDACTED] NI
AV [REDACTED] Cm
c3 [REDACTED] E8
4x [REDACTED] Q==

--- END SourceFire Product License ---
```

Figure 1: Product license

This license actually looks like a PEM encoded certificate, but looking inside undoing the *base64* encoding

00000000:	6d6f 6465 6c20 3078 3432 3b0a 6578 7069	model 0x42;.expir
00000010:	7265 7320 666f 7265 7665 723b 0a6e 6f64	res forever;.nod
00000020:	6520 [REDACTED]	e [REDACTED]
00000030:	[REDACTED] 3b 0a73 6572 6961 6c5f 6e75 6d62	;.serial_num
00000040:	6572 20 [REDACTED] 3b0a 6665	er [REDACTED];.fe
00000050:	6174 7572 655f 6964 2030 7841 3b0a 7365	ature_id 0xA;.se
00000060:	7269 6573 5f33 5f6d 6f64 656c 5f69 6e66	ries_3_model_inf
00000070:	6f20 3633 473a 313a 5052 4f54 4543 542b	o 63G:1:PROTECT+
00000080:	434f 4e54 524f 4c3b 0a36 3347 2033 4437	CONTROL;.63G 3D7
00000090:	3131 303b 0a2d 2d2d 1501 b050 1d6c 1766	110; [REDACTED]P.L.f
000000a0:	37b7 42e1 1b66 24c3 97dc 95e6 89bf 9b38	7.B..f\$.8
000000b0:	1099 23ac d943 f42f 3434 9693 e29b 0b53	..#..C./44....S
000000c0:	b55c 94e4 b7b5 9ab4 b35a 981d 115c d7f9	.\.....Z...\.
000000d0:	7c30 e0b4 e962 683f 371e de68 302a 4efc	l0...bh?7..h0*N.
000000e0:	24ca 313e aaca 0bd5 915f 2f10 3c4d 66db	\$.1>...../.<Mf.

Figure 2: License content

The license is in fact a bunch of metadata regarding the license –red– and a binary signature appended after it –green– the serial number is a integer that doesn't seems to be a timestamp, and the node is the MAC address of the device.

But, how do we request a new license? What data should we provide to request a license? The request process is easy, the only thing we must do is go to the license option in the FSM and click the option for adding a new license:

Add Feature License

License Key **66:11:22:33:44:55:66**

License

If your web browser cannot access the Internet, you must switch to a host with Internet access and navigate to <https://www.cisco.com/go/license>.

Using the license key, **66:11:22:33:44:55:66**, follow the on-screen instructions to generate a license.

Figure 3: License content

Here we can see that the necessary data is the "License Key", and a *PAK* code given in the documentation of the appliance –this code wouldn't be necessary for our purposes–, but, what is actually this code? It is a device type code "66" in hexadecimal `0x42`, do you remember this number from the license file? And the MAC address of the FSM –in case the license is for a FSM appliance–.

Now that we know how a license is built, let's try to load a modified one, and yes, it doesn't work, the console launches a beautiful **Failed [signature]** error.

Once we know this details, let us search around to look for the process that validates the licenses.

3 Checking the validation system

One of the great things about the FireSIGHT systems is that all the appliances are built on top of a custom linux distribution, with active SSH server and root access.

Looking and greping around, points us on a *Perl* module, `/var/sf/lib/perl/5.10.1/SF/System.pm`, this perl module contains all the stuff regarding to FireSIGHT system interactions.

```
sub checklic_parse {
    my %params = validate(@_, {
        source => {
            callbacks => {
                'Type Validator (system.file)' => sub {
                    SF::Types::is_valid('system.file', $_[0])
                },
            },
            optional => 0,
        },
    });

    # Run checklic
    my $result = SF::System::Wrappers::RunCmd([
        SF::Reloc::RelocateFilename($SFBINPATH . '/checklic'),
        '-d',
        '-f',
        SF::Reloc::RelocateFilename($params{source})
    ]);

    # Parse stdout
    my %license = ();
    foreach my $line (split(/\n/, $result->{stdout})) {
        if ($line =~ /\[(\w+):\s(.*)\]/) {
            $license{$1} = $2;
        } elsif ($line eq 'Failed [ date ]') {
            $license{status} = getPkgVar('SF::Auth', '$AUTH_LICENSE_DATE_INVALID');
        } elsif ($line eq 'Failed [ signature ]') {
            $license{status} = getPkgVar('SF::Auth', '$AUTH_LICENSE_SIG_INVALID');
        } elsif ($line eq 'Failed [ node ]') {
            $license{status} = getPkgVar('SF::Auth', '$AUTH_LICENSE_NODE_INVALID');
        } elsif ($line eq 'Failed [ model ]') {
            $license{status} = getPkgVar('SF::Auth', '$AUTH_LICENSE_MODEL_INVALID');
        } elsif ($line eq 'Failed [ unknown ]') {
            $license{status} = getPkgVar('SF::Auth', '$AUTH_LICENSE_OTHER_INVALID');
        } elsif ($line eq 'Valid license') {
            $license{status} = getPkgVar('SF::Auth', '$AUTH_LICENSE_VALID');
        }
    }

    return \%license;
}
```

Figure 4: License parsing function

The important part here seems to be the "Run checklic" line. Looking for the `$SFBINPATH` variable declaration we can locate the called program "checklic" in the path `/usr/local/sf/bin/`.

Now let's see what this program actually is:

```
# file /usr/local/sf/bin/checklic
checklic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.0, stripped
```

Ok, we have here a nice stripped binary executable, let's try to execute it with the flag `-h`:

```
Copyright Sourcefire 2013
-v version
-h this message
-f [filename]
-k [keyfile]
-q quietmode
-d dump contents of license
-F dump feature licenses
```

Bingo! thanks Cisco! running the version command returns:

```
Authentication version 1
```

The perl module, as seen supra, uses the flags `-d` and `-f` passing to it a license file, the output of a valid license is:

```
Valid license
[model: 0x42]
[expires: forever]
[node: 00:11:22:33:44:55]
[serial_number: 123456789]
[feature_id: 0xC]
[model_info: 66E:5000:HOST,66E:5000:USER]
[66E: VirtualDC64bit]
```

The resulting output, is the parsed license information with a "license status" header, as shown in the `checklic_parse` function, we have six different options:

- `AUTH_LICENSE_NOT_CHECKED => 0x00;`
- `AUTH_LICENSE_VALID => (1<<0);`
- `AUTH_LICENSE_DATE_INVALID => (1<<1);`
- `AUTH_LICENSE_SIG_INVALID => (1<<2);`

- `AUTH_LICENSE_NODE_INVALID => (1<<3);`
- `AUTH_LICENSE_MODEL_INVALID => (1<<4);`
- `AUTH_LICENSE_OTHER_INVALID => (1<<7);`

This constants are defined in the perl module `/var/sf/lib/perl/5.10.1/SF/Auth.pm`

4 Bypassing the license validation

The easiest way to bypass the validation system is always returning the `$AUTH_LICENSE_VALID` constant, no matters of the real *checklic* binary output. Ok, it works... but it's a pretty boring and ugly solution.

4.1 The imitation game

Another easy solution is replacing the *checklic* binary by one of our design, one that parses the given license and returns a *Valid license* header.

As expected, after some few lines of code for the parser we accomplished our goal with a nice license validator. The cons of this approach is that **ANY** license will be validated, and this could lead to system malfunctions due to zero license sanity checks.

4.2 A nicer but not so appropriate approach

Let's go a step further, let's look inside the binary to check the validation flow. The first thing to look for is the string `Failed [something]`, this is the output of the license validation failure, after digging a little around this appears:

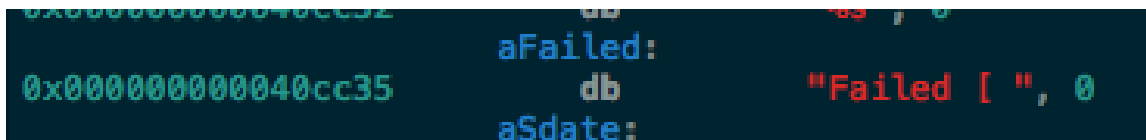


Figure 5: Failed string

This points with a XREF to 0x402a2c and this is part of a procedure that validates the license on address 0x4029da


```

; ===== BEGINNING OF PROCEDURE =====
; Variables:
; var_4: -4
; var_8: -8
; var_C: -12
; var_1010: -4112
; var_1018: -4120

XXXValidateLicense:
0x00000000004029da    push    rbp                                ; End of unwind block (FDE at 0x40f01c), Begin of unwind
0x00000000004029db    mov     rbp, rsp
0x00000000004029de    sub     rsp, 0x1020
0x00000000004029e5    mov     qword [rbp+var_1018], rdi
0x00000000004029ec    mov     dword [rbp+var_4], 0x0
0x00000000004029f3    mov     rax, qword [rbp+var_1018]
0x00000000004029fa    mov     rdi, rax                                ; argument #1 for method sub_403a2e
0x00000000004029fd    call    sub_403a2e                                ; sub_403a2e
0x0000000000402a02    mov     dword [rbp+var_8], eax
0x0000000000402a05    cmp     dword [rbp+var_8], 0x0
0x0000000000402a09    je      loc_402ba8

0x0000000000402a0f    lea     rdx, qword [rbp+var_C]
0x0000000000402a13    mov     rax, qword [rbp+var_1018]
0x0000000000402a1a    mov     rsi, rdx                                ; argument #2 for method sub_404825
0x0000000000402a1d    mov     rdi, rax                                ; argument #1 for method sub_404825
0x0000000000402a20    call    sub_404825                                ; sub_404825
0x0000000000402a25    lea     rax, qword [rbp+var_1010]
0x0000000000402a2c    mov     esi, aFailed                                ; argument "__format" for method j_sprintf, "Failed [ "
0x0000000000402a31    mov     rdi, rax                                ; argument "__s" for method j_sprintf
0x0000000000402a34    mov     eax, 0x0
0x0000000000402a39    call    j_sprintf                                ; sprintf
0x0000000000402a3e    mov     eax, dword [rbp+var_C]
0x0000000000402a41    and     eax, 0x2
0x0000000000402a44    test    eax, eax
0x0000000000402a46    je      loc_402ba8

0x0000000000402a48    lea     rdx, qword [rbp+var_1010]
0x0000000000402a4f    lea     rax, qword [rbp+var_1010]
0x0000000000402a56    mov     esi, aSdate                                ; argument "__format" for method j_sprintf, "%sdate "
0x0000000000402a5b    mov     rdi, rax                                ; argument "__s" for method j_sprintf
0x0000000000402a5e    mov     eax, 0x0
0x0000000000402a63    call    j_sprintf                                ; sprintf

```

Figure 6: License validation procedure

After a fast inspection of the flow, it's easy to deduct that changing the instruction on address 0x402a05 from 0x0 to 0x1 will jump directly over the validation, avoiding the failed status and move the execution straight to address 0x402ba8, getting our wanted Valid license response

```

loc_402ba8:
0x0000000000402ba8    mov     edi, aValidLicensen
0x0000000000402bad    mov     eax, 0x0
0x0000000000402bb2    call    XXXVsprintfToStdOut

loc_402bb7:
0x0000000000402bb7    mov     eax, dword [rbp+var_4]
0x0000000000402bba    leave
0x0000000000402bbb    ret
; endp

```

Figure 7: Valid license loc

Applying the next patchline will do the job:

```
printf '\x01' | dd seek=$((0x2a08)) conv=notrunc bs=1 of=/usr/local/sf/bin/checklic
```

The bad news here are that any original license will stop working due to a nice "Failed [license]" message, this will include the already loaded and validated licenses and the new not yet uploaded ones.

To make both work it will take some more effort patching the binary, modifying multiple parts of `sub_4029da`, but our approach is anyway not too shabby.

Let's try an even more elegant solution.

5 Searching for the signature certificate

One thing is clear, the license file is signed, and where a signature is it must be a valid certificate/signing key. The first thing that I tried was to locate all the `.pem`, `.der` and `.key` files without luck, there are plenty of certificates and keys but none of them is the one we are looking for.

5.1 Intercepting the validation system

If we remember, there is a param for a key on the *checklic* binary, lets try to find when it's passed to the binary replacing the original *checklic* by a wrapper that logs all the calls to the binary –important is to maintain the same permissions and owners–.

The following bash script was used in place of the original *checklic*:

```
#!/bin/sh

### Define or variables
cmdLine=$*
checklicBin="/Volume/home/admin/checklic"
now='date'
parent="$(ps -o comm= $PPID)"
logFile="/tmp/sflicCMDLog.log"
logLine="[${now}] (Parent: $parent) $cmdLine"
evLine="${checklicBin} ${cmdLine}"

### Save the command to or logfile
echo $logLine >> $logFile

### Eval the original command calling or checklic
eval "${checklicBin} ${cmdLine}"
```

The purpose of this script is to log all the calls done against *checklic* into a log file placed under `/tmp/` and execute the original *checklic* with the initial params.

The first thing we notice is that it's called every minute by `Syncd.pl`, asking for a license content dump, the loaded licenses are stored under `/etc/licenses.d/` in files with the *md5* of the license itself with a *.lic* extension.

After a while and after loading a custom license, no key is passed to the binary. Let's look with a *strace* if the application loads it from an hardcoded location with `strace checklic -d -f test.lic` command, the result is also negative, the only loaded files are multiple shared objects from standard libraries, some bus files and a configuration file `–/etc/sf/ims.conf–` that hasn't any clue for our problem.

5.2 Looking inside the guts of *checklic* revisited

Let's recap, the signature certificates/keys/whatever aren't stored on disk –apparently– nor is checked against any remote server through an internet access –our FSM isn't connected to the internet–, so, the only place where this keys could be is embedded into the binary.

After a fast visual inspection of the entropy of the disassembled binary an interesting zone appeared at the bottom right corner

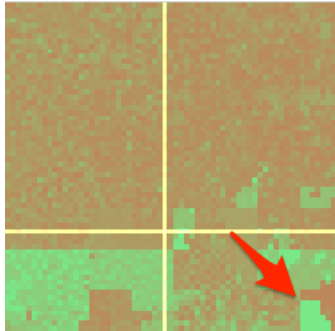


Figure 8: Entropy map

This section contains a big chunk of random data, and the segment has a very descriptive name `gPublicKeys`

```
gPublicKeys:
0x00000000000610b40 db 0x40 ; '@'
0x00000000000610b41 db 0xd8 ; '.'
0x00000000000610b42 db 0x40 ; '@'
0x00000000000610b43 db 0x00 ; '.'
0x00000000000610b44 db 0x00 ; '.'
0x00000000000610b45 db 0x00 ; '.'
0x00000000000610b46 db 0x00 ; '.'
0x00000000000610b47 db 0x00 ; '.'
0x00000000000610b48 db 0x30 ; '0'
0x00000000000610b49 db 0x82 ; '.'
0x00000000000610b4a db 0x02 ; '.'
0x00000000000610b4b db 0x0a ; '.'
0x00000000000610b4c db 0x02 ; '.'
0x00000000000610b4d db 0x82 ; '.'
```

Figure 9: Public keys segment

The segment contains basically two chunks of random data separated by some null bytes. After dumping this two chunks, the first thing that catches our attention is the difference in the size of the first chunk with 449 bytes and the second one with 558 bytes, the other thing is that both chunks end with `0x02`, `0x03`, `0x01`, `0x00`, `0x01` –in red, infra– this is characteristic of an ASN1 DER encoded data structure.

Digging a little bit deeper at the data chunks, it's clearly visible that these chunks have a padding –in green, infra–, and after it, a sequence of the beginning of an ASN1 DER

encoded public RSA key – x30, x82, x02, x0a, x02, x82, x02, x01, x00–, this is the typical and characteristic *MIH* begin of a base64 encoded RSA PEM key

```

\x40\xd8\x40\x00\x00\x00\x00\x00\x30\x82\x02\x0a\x02\x82\x02\x01\x00\xea\xbc\x30\x70\xa9\x1f\x87\x51\x8e\x23\xad\x55\x8f
\x0c\x0c\xcd\x46\xee\x3e\x48\xff\x7b\xd7\x8a\x2e\x87\x5b\x46\x71\xbb\x47\xa8\xfe\x4f\xb6\x2c\x02\x37\xc5\x07\x7c\x8c\xb7
\x5f\xaf\xbf\x18\x58\xc0\xe5\x2a\x18\xe7\x48\xb1\x25\xe0\x52\x20\xbc\xf1\x55\x76\x3b\x7f\xeb\xc8\x8f\xd1\x5f\x76\xd8\x25
\xbd\xef\x08\xa6\xe6\xca\x3e\x75\x23\x85\x64\x48\xae\xe9\x67\x71\xbe\x83\xf3\x5f\x0e\xb8\x10\xfb\x2c\x99\x33\x0c\x4a\x40
\x52\xb8\x07\xb6\xc1\xdd\x7f\x4a\x71\xc3\xf5\x1b\x23\xa3\x9f\x57\xa5\x11\xdf\xd7\xab\x17\xa4\x5a\x04\x61\xda\x7e\x9e\x18
\x71\x39\x97\xb0\x81\xbb\xed\xdf\x5d\x4c\xa9\x93\xb7\x92\x8c\xde\x83\x94\x9c\x5e\x2c\x22\xca\x8b\x48\xeb\x70\x4c\x02\x40
\x7e\x04\x65\xd0\x9c\x18\x64\x76\x00\xbb\x96\x43\xa7\xcf\x37\xd2\xb1\xae\xde\xc0\x74\x71\x35\x8f\xf6\xf0\x41\xed\xb1\xa9
\x73\xae\x05\xd2\x2c\x55\x65\x69\x11\x37\xbe\x14\xf6\x16\x95\x59\x94\xb1\x99\x89\xb1\xf9\x19\xf8\x92\x26\x73\x15\xf9\x53
\xbc\x5e\x0f\x87\x46\x8e\x4b\x79\xf5\x11\x6c\xf6\xca\x43\x25\x70\xa9\x94\x6d\xe7\xe5\xb7\xef\x7e\xef\x9c\xb8\xb5\xff\xa3
\x04\x3e\x34\x93\x43\x39\x6c\x6c\x1a\x3c\x7b\xf6\x33\x25\x72\x51\x49\x4d\x41\x5d\xf2\xad\xb5\x4d\xcc\x83\x41\x01\xcc\x1b\x4
\x5e\x52\x7e\x62\x58\xe2\x3\xac\xbd\xce\xda\xfa\x6f\x71\xe5\x22\x15\xd8\xb9\x61\x82\x53\x7d\x89\x4e\xde\x60\x3d\x20\x9b
\x7f\x97\x1c\x00\x9f\x0b\x15\xce\xff\x68\x3e\x61\x80\xb7\xaa\x46\x0b\x2a\xda\x4d\xf3\xb2\x56\xb3\x18\xf4\xe8\xb3\x56\x71
\x6e\x23\xf6\x5a\x27\xa1\x29\x98\x9c\x7e\x0f\xde\x16\x34\x0b\x72\x6d\xf1\x80\xa4\x11\x4d\x3d\x77\xf1\x6e\x0c\x48\x16\xd0
\x89\x8e\x33\x17\xf7\xf3\xf8\x26\x9f\x9d\x37\x77\xab\x15\xcf\x28\x9a\x59\x45\x4e\x43\x60\x4c\x90\x37\x72\xe0\x2c\x22\x24
\x49\xab\x6e\xbd\x8d\xdd\xbc\x50\x0e\x69\x27\x9d\x5a\x97\x6e\xcc\x23\x93\x5d\x48\xe5\x5c\xe7\x3d\x1b\x0b\x95\xf0\xbb\xac\x9e
\x3e\x4b\xaf\x58\x62\x78\xfc\x54\x43\x0b\x8a\x5d\x00\x31\x24\xda\x69\xbe\x9f\x48\x30\x5c\xdc\x99\x25\xca\x88\x3d\x86
\x09\xaa\xe8\x5d\x90\xbf\xfa\xbd\x21\xb0\x6c\x06\x00\xda\x77\x55\x5c\x88\x3e\x7e\xf7\xce\x41\x22\x02\xe1\x23\x56\xd2\x8a
\x59\x98\xbe\x9e\x05\xbd\x4c\x5c\x31\xa1\xe2\x26\xbd\x06\x47\xe2\x70\x25\x59\x55\x02\x03\x01\x00\x01

```

Figure 10: RSA key one

```

\x23\x21\xe9\x2d\x7d\xa2\x4a\x5f\xd8\x5d\x94\xee\x21\x71\x3e\xb4\x3f\x51\x1a\xf2\x0e\x02\x00\x00\x4b\xd8\x40\x00\x00\x00
\x00\x00\x30\x82\x02\x0a\x02\x82\x02\x01\x00\xff\x9e\x2b\x8c\x76\xf9\x16\xdf\x78\x7c\x80\x8b\xed\x41\xec\xa0\x58\x6f\x7f
\x96\xa8\xd0\x66\x2e\x19\xe7\xcf\xf5\x41\x76\x0e\x52\xf0\xa2\xc1\x73\x30\xd1\x0c\x52\x6a\x56\xd8\xf4\x6d\x31\x57\x47\x84
\x5f\x55\x31\xa6\x7d\xb2\xa8\xe3\xe4\xa0\x9a\x3a\x8d\x1d\x00\xa9\x1e\xb2\xf3\x48\x9c\x0e\x91\xde\x47\x2e\xfd\x73\xcb\xe9
\x4e\x9b\x51\x35\x2a\x69\x99\x37\x66\x84\xf7\x8f\x76\x2f\x01\x76\x25\xd8\x43\x6c\x9a\xd1\x20\x92\xe2\x75\x5a\x28\x36\xa5
\x85\x2e\x88\x8f\xff\xdd\x67\xd9\x0e\x86\x9a\xa4\x23\xf6\x7d\x64\xd3\x96\x30\x7d\xd3\x05\x44\x6c\x8c\x38\x05\x72\x00\x70
\x87\xc0\x25\x53\x59\x5a\xe0\xf3\x06\xd0\xa2\x40\xac\x26\x5e\xf3\xf0\x49\x2b\x9c\x6a\x0b\x21\x5b\x1b\x43\x29\x06\xbe\xda
\x5c\x4d\xdf\x66\xa0\x42\x65\x9d\x01\xe5\xce\x08\x39\xad\x89\x15\xbb\x30\x7f\x4f\x78\x5c\x83\xa2\x93\xdc\xf9\x27\x55\xe9
\x48\x97\xf9\x1b\x3c\x03\xf8\x69\xf0\xb4\xe8\x02\x10\xe9\x69\xf5\x20\x70\x34\x5f\x1d\x5a\x94\x2e\xf9\xb0\x45\x31\x87\x73
\x11\x93\x68\x02\x27\xe0\x1d\x17\x09\xff\xab\xa6\x85\x68\x02\x78\x0e\x68\x66\x9b\x37\x00\x05\x7e\x82\x05\x7c\xdb\xe6\xee
\x17\x3c\xcd\x01\x0d\xa5\x0e\x89\x3f\x9c\x89\x43\x7c\x83\xf0\x8c\xf8\x42\x0a\x79\xa7\x1f\xbd\x29\x33\xb8\x00\xe6\x00\xa6
\x45\x76\xe9\x9f\x02\xde\x04\x06\x53\xd9\x0d\x27\x0a\x0e\x74\xf6\x86\x76\x5f\x04\x9e\x42\x9c\x54\x7c\xae\x90\x0c\x06\xda
\x46\x52\x13\x8b\xa2\x13\x57\x7c\xbb\xe9\x17\x9d\x5a\x37\xad\x0b\x26\x6e\x95\x09\x97\xfb\x78\x0c\x7d\x26\x83\x8f\x20\xa5
\xe3\x98\xa3\x6c\x3f\x47\x38\x7f\x34\x02\x19\x22\x0c\x25\x18\xb2\x80\xd3\x3d\x73\xa4\x5b\x03\x14\x90\xeb\x30\xa0\xf4\x2c
\xfb\x0b\x22\x23\x6f\x07\x6c\x00\x40\xec\x11\xa9\x2b\x28\xf1\x04\x7f\x37\x54\xf8\x0e\xeb\x8d\x0e\x21\xf4\xad\x4e\x7c\x20
\xfe\x92\x07\xff\x5c\x7e\x0f\xdb\x67\x03\x48\x57\x52\x8c\x06\xf2\x71\x21\x0a\xe7\x78\xcf\x51\x1f\x21\x73\x80\xa1\x7c\x46
\xea\xa4\xbd\x97\x54\x89\x78\x3a\xbc\x8f\x05\x4c\x8c\x21\x0b\x8d\xe3\x09\x87\x38\xdd\x51\x2e\x31\x6b\xfe\x53\x73\x7c\x67
\x8e\x7d\xa6\xa3\x88\x3f\xb5\x27\x7c\x7f\x8b\x00\x2a\xfc\x39\x09\x8e\xce\x4b\x9d\x63\xef\x7c\x02\x7a\xaf\x5c\x9d\x20\xa4
\xa0\x15\xf4\x5e\x04\xcc\x45\x28\x00\x43\x4d\x0a\x75\x02\x03\x01\x00\x01

```

Figure 11: RSA key two

The procedure responsible for loading the key from memory and convert it to a *RSA-PublicKey* structure is `sub_406482`.

Taking a look at it we can see that the *SHA1* digest of the certificate is calculated and compared to something placed at the `offset + 0x408` of the actual keys bytes chunk position and taking `0x14` bytes from it, yes, 20 bytes, the length of a *SHA1* digest!

```

0x00000000000406482      push     rbp
0x00000000000406483      mov      rbp, rsp
0x00000000000406486      sub      rsp, 0x470
0x0000000000040648d      mov      dword [rbp+var_464], edi
0x00000000000406493      mov      rdx, qword [qword_6105f0]
0x0000000000040649a      mov      eax, dword [rbp+var_464]
0x000000000004064a0      cdqe
0x000000000004064a2      shl      rcx, 0x5
0x000000000004064a6      mov      rcx, rcx
0x000000000004064a9      shl      rcx, 0x5
0x000000000004064ad      add      rcx, rcx
0x000000000004064b0      add      rax, rdx
0x000000000004064b3      lea      rdx, qword [rbp+var_430]
0x000000000004064ba      mov      rsi, rax
0x000000000004064bd      mov      eax, 0xB4
0x000000000004064c2      mov      rdi, rdx
0x000000000004064c5      mov      rcx, rax
0x000000000004064c8      rep movsq qword [rdi], qword [rsi]
0x000000000004064cb      mov      eax, dword [rbp+var_14]
0x000000000004064ce      cdqe
0x000000000004064d0      lea      rdx, qword [rbp+var_450]
0x000000000004064d7      lea      rcx, qword [rax+0x408]
0x000000000004064de      add      rcx, 0x8
0x000000000004064e2      mov      rsi, rax
0x000000000004064e5      mov      rdi, rcx
0x000000000004064e8      call     j_SHA1
0x000000000004064ed      lea      rax, qword [rbp+var_430]
0x000000000004064f4      lea      rcx, qword [rax+0x408]
0x000000000004064fb      lea      rax, qword [rbp+var_450]
0x00000000000406502      mov      edx, 0x14
0x00000000000406507      mov      rsi, rcx
0x0000000000040650a      mov      rdi, rax
0x0000000000040650d      call     j_memcmp
0x00000000000406512      test     eax, eax
0x00000000000406514      je       loc_406560

```

Figure 12: Load public key

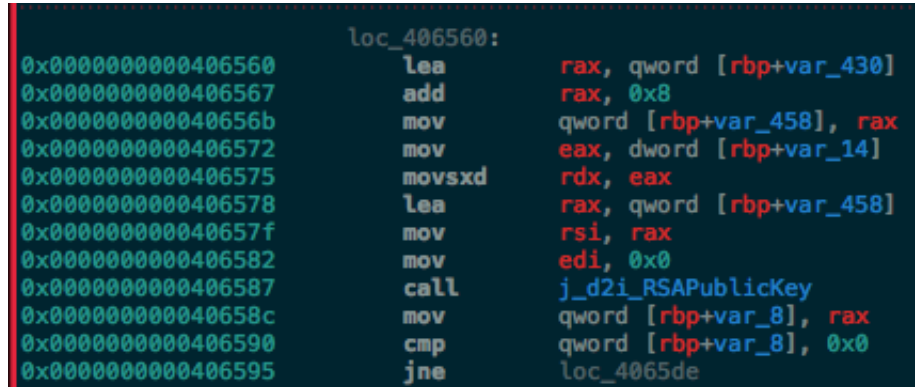
With this in mind we can teardown our extracted bytes chunk to a more coherent structure

[illegible]

Figure 13: Cert structure teardown

Now we have an 8 byte long padding, a 526 byte long PKCS#1 RSA public Key, 498 null bytes, a 20 bytes *SHA1* digest –presumable of the RSA key– and a 4 bytes trail.

The PKCS#1 is decoded to a *RSAPublicKey* on `loc_406560`



```

loc_406560:
0x0000000000406560      lea     rax, qword [rbp+var_430]
0x0000000000406567      add     rax, 0x8
0x000000000040656b      mov     qword [rbp+var_458], rax
0x0000000000406572      mov     eax, dword [rbp+var_14]
0x0000000000406575      movsxd  rdx, eax
0x0000000000406578      lea     rax, qword [rbp+var_458]
0x000000000040657f      mov     rsi, rax
0x0000000000406582      mov     edi, 0x0
0x0000000000406587      call    j_d2i_RSAPublicKey
0x000000000040658c      mov     qword [rbp+var_8], rax
0x0000000000406590      cmp     qword [rbp+var_8], 0x0
0x0000000000406595      jne     loc_4065de

```

Figure 14: PKCS#1 decode

Now that we know the exact offsets of the certificate components, let's dump the embedded RSA key.

```
dd skip=$((0x10b48)) conv=notrunc bs=1 count=526 if=checklic of=cert1_dump.der
```

This will dump the first certificate to a file, and after calculating the *SHA1* digest of it, bingo!

```
# shasum -a1 cert1_dump.der
2321e92d7da24a5fd85d94ee21713eb43f511af2
```

That's the same one of the data chunk.

5.3 Patching the binary with our keys

The first try was to generate a RSA key pair of 4096 bits with *OpenSSL*, but it didn't work, my second try was to create my own RSA Key generator with *go* –the code is available on my git–.

Once generated the key pair I prepared the `public.der` file for a proper patchline with

```
hexdump -ve '1/1 "_x%.2x"' public.der | sed 's/_/\//g'; echo ""
```

the second step was calculate the *SHA1* and formatting it the same way.

With the public key and the hash, the composition of our patchlines is easy as:

```
printf '\x30\x82\x02\x0a[...] \x03\x01\x00\x01' | dd seek=$((0x10b48)) \
conv=notrunc bs=1 of=checklic_patched
```

For the key, and:

```
printf '\x3A[...] \xE8' | dd seek=$((0x10f48)) conv=notrunc bs=1 \
of=checklic_patched
```

For the hash, and this is it, now, our `checklic` binary is patched for our own RSA keys, this patch must be applied to any device that will make use of a custom signed license, this is for every sensor and FSM.

The next step will be create our own license generator!

6 Building our own license generator

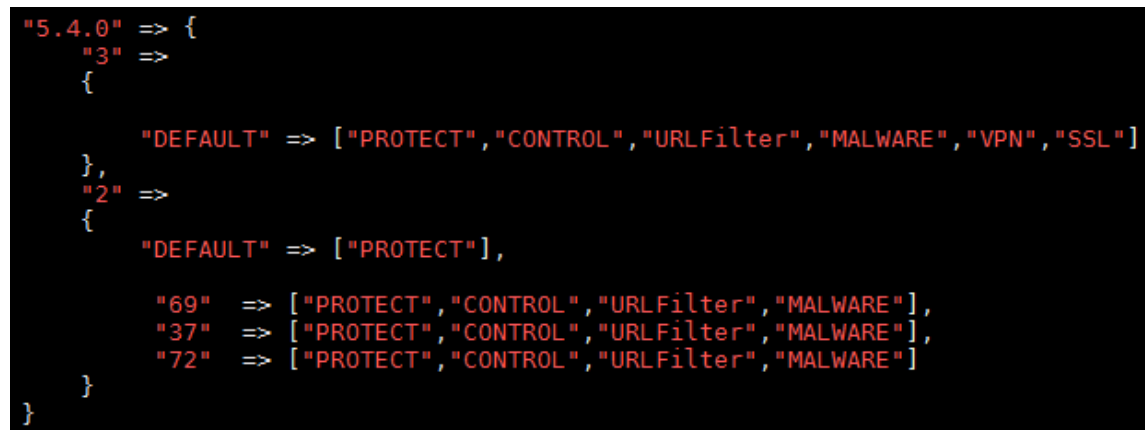
Actually we know how a license looks like:

```
model 0x42;
expires forever;
node 00:11:22:33:44:55;
serial_number 123456789;
feature_id 0xC;
model_info 66E:50000:HOST,66E:50000:USER;
66E VirtualDC64bit;
---${signature}
```

The most important things we need to know here is what the `feature_id` and the `model` codes actually are, after *grep*ing and *finding* a little around I came against these perl modules:

- `/var/sf/lib/perl/5.10.1/SF/LicenseCaps.pm`
- `/var/sf/lib/perl/5.10.1/SF/Sensor.pm`

At the beginning of the *LicenseCaps.pm* file we can find the allowed capabilities map, this will be perfect for defining our sensor license capabilities.



```
"5.4.0" => {
  "3" =>
  {
    "DEFAULT" => ["PROTECT", "CONTROL", "URLFilter", "MALWARE", "VPN", "SSL"],
  },
  "2" =>
  {
    "DEFAULT" => ["PROTECT"],
    "69" => ["PROTECT", "CONTROL", "URLFilter", "MALWARE"],
    "37" => ["PROTECT", "CONTROL", "URLFilter", "MALWARE"],
    "72" => ["PROTECT", "CONTROL", "URLFilter", "MALWARE"]
  }
}
```

Figure 15: License allowed capabilities

In our case we will use the full suite of capabilities, it's free! except the **MALWARE** one, this feature makes use of Cisco's Cloud, and I hope that there are more controls on the Cloud side –further tests will confirm this–, so, our sensor capabilities line will end as:

```
[series_3_model_info: MMM:N:PROTECT+CONTROL+VPN+SSL]
```

For the basic license and other one with a *subscription* type license for the **URLFilter** capability. This kind of licenses are a little different from the *capability* licenses:

```
model 0x42;
expires forever;
node 00:11:22:33:44:55;
serial_number 123456789;
feature_id 0xB;
model_info MMM:N:URLFilter;
MMM 3DXXXX;
license_type SUBSCRIPTION;
```

Being MMM the model info, N the number of licenses –one for each sensor– and XXXX the model name.

The next interesting information is around line 248, where the following variables are defined:

- \$FEATURE_CODE_SERIES3 = 0x0A;
- \$FEATURE_CODE_URLFILTER = 0x0B;
- \$FEATURE_CODE_FIRESIGHT = 0x0C;
- \$FEATURE_CODE_VIRTUAL = 0x09;
- \$FEATURE_CODE_XBEAM = 0x06;

These variables are defining the **feature_id**. In our case, the useful ones are 0xA for Series 3 sensors and 0xC for FSM appliances.

On the *Sensor.pm* the only useful thing we can find for our purposes, are the following definitions

```
our @BIVIO_MODELS = ("38", "46");
our @IS_MODELS = ("36", "37", "42", "54", "69", "72");
our @NORTEL_MODELS = ("41");
our @IS_MODEL_SLICENSED = ();
our @ISSW_MODELS = qw(37 42);
our @VSMODELS = qw(54 69 72);
our @S3_MODELS = qw(60 61 62 63 64 65);

our @FREE_SSL_MODELS = qw(60 61 62 63 64 65); #the same as series 3
```

Figure 16: Sensor models definitions

```

if (SF::Global::getSeriesNumber() == 3)
{
    # Determine the number of network module slots to look for
    my $ims_conf = SF::Util::load_ims_conf();
    my $slotCount = 0;
    given ($ims_conf->{MODELNUMBER}.$ims_conf->{MODELID}) {
        when (/63[BDEQR]/) {
            # Pegasus 1U
            $slotCount = 3;
        } when (/63[CMNS]/) {
            # Pegasus 2U
            $slotCount = 7;
        } when (/63[FGHIJKLP0]/) {
            # Geryon / Chrysaor
            $slotCount = 1;
        }
    }
}

```

Figure 17: Sensor network modules definitions

But this is not really interesting for our needs.

The only useful thing I found for determining the device model ID is, after a correct device pairing, sshing again into the FSM and query the database directly

```
mysql -u root -padmin sfsnort -e "SELECT ip,model_number,model_id FROM sensor;"
```

ip	model_number	model_id
10.0.1.5	66	E
10.0.1.14	63	E

This will be our model number for a *3D8120* sensor; 63E.

And this tied all together is the metadata of our license, this all is followed by a separation of three dashes --- and appending the signature done with our private RSA key in DER format of all the metadata except the three dashes. This process will be clearer after a look on the code available on the aforementioned git repository.

7 Cracking guide

Do this at your own risk!

Cracking software is ilegal and unmoral, please use this only for testing and educational purposes

7.1 RSA key generation

For the generation of our own RSA keys that later will sign our licenses execute:

```
go run rsaGen.go
```

This will generate a public and a private RSA key of 4096 bits. To generate the proper patchline, the public key must be formatted with:

```
hexdump -ve '1/1 "_x%.2x"' public.der |sed 's/_/\x/g'
```

7.2 Calculate the SHA1 of the public key

To calculate and format the RSA public key SHA1 hash on MacOS, use:

```
shasum -a1 crypto/public.der |cut -d" " -f1 | \  
sed -E 's/(.{2})/\1\x/g' |rev |cut -d"\\" -f2- |rev
```

7.3 Prepare the patchlines (x86-64 version)

The following commands will be responsible for patching the binary, only for 64 bits version, for the 32 bits versions (Series 2) the procedure is the same, the only thing that changes is the patch address.

The first one will write our new generated RSA key to the binary and the second line will write the SHA1 hash to the binary.

Make a security copy of the *checklic* binary before this

```
printf '${your_formated_public_key}' | \  
dd seek=$((0x10b48)) conv=notrunc bs=1 of=${target}  
  
printf '${your_formated_sha1}' | \  
dd seek=$((0x10f48)) conv=notrunc bs=1 of=${target}
```

7.4 Patch!

SSH into the FSM and get sudo, execute the two patchlines changing the `${target}` for the checklic binary.

Repeat the same process on all your Sensors.

7.5 Generate your licenses

For a FSM license run:


```
go run sflicgen.go -l 66:00:11:22:33:44:55 \  
-k ../crypto/private.pem -fsm
```

and for a sensor license run:



```
go run sflicgen.go -l 66:00:11:22:33:44:55 \  
-k ../crypto/private.pem -n 6 -mid 63E -mod 3D8120
```

Register them on your FSM, assign the capabilities to your sensors and enjoy.

3D7110

License Type	Status	Number of Licenses	Expires	
Protection Control	Valid License	6	Never (IPS Term Subscription is still required for IPS)	

3D8120

License Type	Status	Number of Licenses	Expires	
Protection Control VPN SSL	Valid License	6	Never (IPS Term Subscription is still required for IPS)	
URL Filtering	Valid License	6	Never (IPS Term Subscription is still required for IPS)	

VirtualDC64bit


License Type	Status	Number of Licenses	Expires	
FireSIGHT Host FireSIGHT User	Valid License	50000 50000	Never (IPS Term Subscription is still required for IPS)	

Figure 18: Uploaded licenses

8 Conclusions

Protecting such a device from reverse engineering is extremely painful or impossible at all, especially if the designer wants that the final user enjoys some "open device" experience.

There are some alternatives to remediate this:

One alternative is closing the appliance to a point that it will be converted into a blackbox, only accessible to the technical support team, making the final user think about the actual risks of letting inside its infrastructure a closed piece of hardware that nobody knows what it actually does.

Another, more elegant solution, could be to add an integrity check on the `kernel` or `initramfs` for detecting modifications on critical system components, this is not a hundred percent effective solution, but it could increase the difficulty of this kind of hacks enormously.