Automation in Reverse Engineering: String Decryption

synthesis.to/2021/06/30/automating_string_decryption.html

Automation plays a crucial rule in reverse engineering, no matter whether we search for vulnerabilities in software, analyze malware or remove obfuscated layers from code. Once we manually identify repeating patterns, we try to automate the process as far as possible. For automation, it often doesn't matter if you use Binary Ninja, IDA Pro or Ghidra, as long as you have the knowledge *how* to realize it in your tool of choice. As you will see, you don't have to be an expert to automate tedious reverse engineering tasks; sometimes it just takes a few lines of code to improve your understanding a lot.

Today, we take a closer look at this process and automate the decryption of strings for a malware sample from the Mirai botnet. Mirai is a malware family that hijacks embedded systems such as IP cameras or home routers by scanning for devices that accept default login credentials. To impede analysis, Mirai samples store those credentials in an encoded form and decode them at runtime using a simple XOR with a constant. In the following, we first manually analyze the string obfuscation. Afterward, we use Binary Ninja's high-level intermediate language (HLIL) API to get all string references and decrypt them.

In static malware analysis, one of the first things to do is to have a closer look at the identified strings, since they often reveal a lot of context. In this sample, however, we mostly see strings like `PMMV`, `CFOKL`, `QWRRMPV` and others. At first glance, they don't make much sense. However, if we have a closer look at how they are used in the code, we notice something interesting: They are repeatedly used as function parameters for the function `sub_10778`. (The corresponding function calls can be found here in the leaked source code.)

```
sub_10778("PMMV", &data_1616c, 0xa)
sub_10778("PMMV", "TKXZT", 9)
sub_10778("PMMV", "CFOKL", 8)
sub_10778("CFOKL", "CFOKL", 7)
sub_10778("PMMV", &data_16184, 6)
sub_10778("PMMV", "ZOJFKRA", 5)
sub_10778("PMMV", "FGDCWNV", 5)
sub_10778("PMMV", 0x1619c, 5)  {"HWCLVGAJ"}
sub_10778("PMMV", &data_161a8, 5)
sub_10778("PMMV", &data_161b0, 5)
sub_10778("QWRRMPV", "QWRRMPV", 5)
```

```
sub_10778("root", "xc3511", 0xa)
sub_10778("root", "vizxv", 9)
sub_10778("root", "admin", 8)
sub_10778("admin", "admin", 7)
sub_10778("root", "888888", 6)
sub_10778("root", "xmhdipc", 5)
sub_10778("root", "default", 5)
sub_10778("root", 0x1619c, 5)  {"juantech"}
sub_10778("root", "123456", 5)
sub_10778("root", "54321", 5)
sub_10778("support", "support", 5)

def decrypt(address, already_decrypted):
    # walk over string bytes until termination
    while True:
        # read a single byte from database
        encrypted_byte = bv.read(address, 1)

        # return if null byte or already decrypted
        if encrypted_byte == b'\x00' or address in already_decrypted:
            return

        # decrypt byte
        decrypted_byte = chr(int(encrypted_byte[0]) ^ 0x22)

        # write decrypted byte to database
        bv.write(address, decrypted_byte)

        # add to set of decrypted addresses
        already_decrypted.add(address)

        # increment address
        address += 1
```

```
# get function instance of target function
target_function = bv.get_function_at(0x10778)
# set of already decrypted bytes
already_decrypted = set()

# 1: walk over all callers
for caller_function in set(target_function.callers):

    # 2: walk over high-level IL instructions
    for instruction in caller_function.hlil.instructions:

        # 3: if IL instruction is a call
        #    and call goes to target function
        if (instruction.operation == HighLevelILOperation.HLIL_CALL and
            instruction.dest.constant == target_function.start):

            # 4: fetch pointer to encrypted strings
            p1 = instruction.params[0]
            p2 = instruction.params[1]

            # 5: decrypt strings
            decrypt(p1.value.value, already_decrypted)
            decrypt(p2.value.value, already_decrypted)
```

Based on this, we can assume that the passed strings are decoded and further processed in the called function. If we inspect the decompiled code of the function, we identify the following snippet that operates on the first function parameter `arg1`. For the second parameter `arg2`, we can find a similar snippet.

```
uint32_t r0_3 = sub_12c90(arg1)
void* r0_5 = sub_14100(r0_3 + 1)
sub_12d0c(r0_5, arg1, r0_3 + 1)
if (r0_3 s> 0) {
    char* r2_3 = nullptr
    do {
        *(r2_3 + r0_5) = *(r2_3 + r0_5) ^ 0x22
        r2_3 = &r2_3[1]
    } while (r0_3 != r2_3)
}
```

The code first performs some function calls using `arg1`, goes into a loop and increments a counter until the condition `r0_3 != r2_3` no longer holds. Within the loop, we notice an XOR operation `*(r2_3 + r0_5) ^ 0x22`, where `*(r2_3 + r0_5)` seems to be an array-like memory access that is xored with the constant `0x22`. After performing a deeper analysis, we can clean up the code by assigning some reasonable variable and function names.

```
uint32_t length = strlen(arg1)
void* ptr = malloc(length + 1)
strcpy(ptr, arg1, length + 1)
if (length s> 0) {
    char* index = nullptr
    do {
        *(index + ptr) = *(index + ptr) ^ 0x22
        index = &index[1]
    } while (length != index)
}
```

Now, we have a better understanding of what the code does: It first calculates the length of the provided string, allocates memory for a new string and copies the encrypted string into the allocated buffer. Afterward, it walks over the copied string and decrypts it bytewise by xoring each byte with `0x22`. This is also in line with the decryption routine of the original source code.

In other words, strings are encoded using a bytewise XOR with the constant value `0x22`. If we want to decode the string `PMMV` in Python, we can do this with the following one-liner.

We walk over each byte of the string, get its corresponding ASCII value via `ord`, xor it with `0x22` and transform it back into a character using `chr`. In a final step, we join all characters into a single string.

After we manually analyzed how strings can be decrypted, we will now automate this with Binary Ninja.

To automate the decryption, we first have to find a way to identify all encoded strings. In particular, we have to know where they start and where they end; in other words, we aim to identify all encrypted bytes. In the second step, we can decrypt each byte individually.

Beforehand, we noticed that the encoded strings are passed as the first two parameters to the function `sub_10778`. To obtain the encoded strings, we can exploit this characteristic by searching for all function calls and parse all passed parameters. Using Binary Ninja's high-level intermediate language (HLIL) API, we can realize this within a few lines of code.

After fetching the function object of the targeted function `sub_10778`, we walk over all functions calling `sub_10778`. For each of these calling functions (referred to as *callers*), we need to identify the instruction that performs the call to `sub_10778`. In order to do this, we walk over the caller's HLIL instructions; for each instruction, we then check if its operation is a call and if the call destination is the targeted function. If so, we access its first two parameters (the pointers to the encoded strings) and pass them to the decryption function. Since some strings—such as `PMMV` —are used as parameters multiple times, we ensure that we only decrypt them once. Therefore, we collect the addresses of all bytes that we already have decrypted in a set called `already_decrypted`.

Up until now, we identified all parameters that flow into the decryption routine. The only thing left to do is to identify all encrypted bytes and decrypt them. Since each parameter is a pointer to a string, we can consider it as the string's start address. Similarly, we can determine the string's end by scanning for terminating null bytes.

Taking the string's start address as input, we sequentially walk over the string until we reach a byte that terminates the string or that was already decrypted. For each byte, we then transform it into an integer, xor it with `0x22`, encode it as a character and write it back to the database. Afterward, we add the current address to the set `already_decrypted` and increment the address.

Finally, we have all parts together: We walk over all function calls of the string decryption function, parse the parameters for each call and decrypt all the strings in Binary Ninja's database. If we put everything into a Python script and execute it, the decompiled code from above contains all strings in plain text.

Automation allows us to spend less time with tedious and repetitive reverse engineering tasks. In this post, I tried to emphasize the thought process behind automation on the example of decrypting strings in malware. Starting with manual analysis, we first pinpointed interesting behavior: encrypted strings used as function parameters. Then, we put it into context by digging into the function, and learned that the strings are decrypted inside. By noticing a recurring pattern—that the function is called several times with different parameters—we developed an idea of how to automate the decryption. By using Binary Ninja's decompiler API, we walked over all relevant function calls, parsed the parameters and decrypted the strings. In the end, 20 lines of code sufficed to improve the decompilation and achieve a much better understanding of the malware sample.

Even if you are just starting out, I encourage you to get familiar with the API that your tool of choice exposes, and to automate some of the tedious tasks you encounter during your day-to-day reversing. It is not only fun; reverse engineering also becomes so much easier.

For questions, feel free to reach out via Twitter @mr_phrazer, mail tim@blazytko.to or various other channels.