

Software Assurance Tips

A product of the Software Assurance Tips Team[1]

Jon Hood

Monday 2nd August, 2021

1 Stripping: An Inefficient Obfuscation Technique

Updated Monday 30th August, 2021

Recently, we were given a piece of software with special handling instructions. The software contained a function which was supposed to be protected: enemies, competitors, and no one without a need-to-know was to ever see how this function manipulated the parameters it was given.

While reviewing the security of the software, we discovered that the developer compiled the binary and released it on their website. When asked how they were able to release such a private routine publicly, the customer claimed that it was fine to release in binary form. Supposedly, the compilation method they used removed the “context of the Human-Readable Source Code used to generate the Machine-Readable Object Code from propagating into the Machine-Readable Object Code.”

Let’s put this claim to the test! For the sake of creating a fully unclassified example, suppose that no one has ever created a function for calculating factorials, and a new intern fresh out of college submits the code in Listing 1 to solve this highly-secretive, important function.

```
#include <stdio.h>
#include <stdlib.h>
void factorial(int argc, char *argv[])
{ //SUPER SEKRIT FACTORIALS
    unsigned long long ret = 1;
    int maxVal = atoi(argv[1]); //0<=maxVal<=20
    for (int i = 1; i <= maxVal; i++)
        ret = ret * (unsigned long long)i;
    printf("%llu\n", ret);
}
```

Listing 1: Unsafe Factorial Function

When compiled into machine code, the function isn’t nearly as easy to follow. Figures 2 and 3 show the unstripped and stripped functional machine code respectively. Had the software been compiled in debug mode, the source code would have been included alongside the machine code.

```

push  %rbp
mov   %rsp,%rbp
sub   $0x30,%rsp
mov   %ecx,0x10(%rbp)
mov   %rdx,0x18(%rbp)
movl  $0x1,0x4(%rbp)
mov   0x18(%rbp),%rax
add   $0x8,%rax
mov   (%rax),%rax
mov   %rax,%rcx
call  29 <factorial+0x29>
mov   %eax,0xc(%rbp)
movl  $0x1,0x8(%rbp)
jmp   45 <factorial+0x45>
mov   0x8(%rbp),%eax
mov   0x4(%rbp),%edx
imul  %edx,%eax
mov   %eax,0x4(%rbp)
addl  $0x1,0x8(%rbp)
mov   0x8(%rbp),%eax
cmp   0xc(%rbp),%eax
jle   35 <factorial+0x35>
mov   0x4(%rbp),%eax
mov   %eax,%edx
lea   0x0(%rip),%rax # 59 <factorial+0x59>
mov   %rax,%rcx
call  61 <factorial+0x61>
nop
add   $0x30,%rsp
pop   %rbp

```

Listing 2: Unstripped Machine Code

```

push  %rbp
mov   %rsp,%rbp
sub   $0x30,%rsp
mov   %ecx,0x10(%rbp)
mov   %rdx,0x18(%rbp)
movl  $0x1,0x4(%rbp)
mov   0x18(%rbp),%rax
add   $0x8,%rax
mov   (%rax),%rax
mov   %rax,%rcx
call  0x29
mov   %eax,0xc(%rbp)
movl  $0x1,0x8(%rbp)
jmp   0x45
mov   0x8(%rbp),%eax
mov   0x4(%rbp),%edx
imul  %edx,%eax
mov   %eax,0x4(%rbp)
addl  $0x1,0x8(%rbp)
mov   0x8(%rbp),%eax
cmp   0xc(%rbp),%eax
jle   0x35
mov   0x4(%rbp),%eax
mov   %eax,%edx
lea   0x0(%rip),%rax # 0x59
mov   %rax,%rcx
call  0x61
nop
add   $0x30,%rsp
pop   %rbp

```

Listing 3: Stripped Machine Code

As can be seen by the stripped vs. unstripped comparison, there is very little (other than the function name) that is different. In fact, once this code is sent through a decompiler (using Binary Ninja), the decompiled code can be seen in figures 4 and 5.

```

int64_t factorial(int32_t arg1, void* arg2)
{
    int32_t var_c = 1;
    int32_t rax_3 = atoi(*(arg2 + 8));
    for (int32_t var_10 = 1; var_10 s
        <= rax_3; var_10 = var_10 + 1)
        var_c = var_10 * var_c;
    return printf(._rdata, zx.q(var_c));
}

```

Listing 4: Unstripped Decompilation with Binary Ninja

```

int64_t sub_100401080(int32_t arg1, void* arg2)
{
    int32_t var_c = 1;
    int32_t rax_3 = atoi(*(arg2 + 8));
    for (int32_t var_10 = 1; var_10 s
        <= rax_3; var_10 = var_10 + 1)
        var_c = var_10 * var_c;
    return printf(data_100403000, zx.q(var_c));
}

```

Listing 5: Stripped Decompilation with Binary Ninja

While compilation and obfuscation definitely make it more difficult to glean the original meaning of software, it's not impossible to trace through the decompilation and figure out the original intent of the developer. If source code is protected because of what it does, the binary generated from that source code should probably be handled with the same protections.

References

- [1] Jon Hood, ed. SwATips. <https://www.SwATips.com/>.