CS 88: Security and Privacy

03: Software Security – Buffer Overflow Attacks

01-30-2024
Announcements

• Clicker registrations posted – let me know if I don’t have yours
• Please choose partnerships for Lab 1 (EdStem) – last chance.
• Reading quizzes count from this week
• Lab 0 is due today
• Midterm dates on edstem later today
Reading Quiz
Today

• What is software security
• CS 31 Recap:
  • functions and the stack
  • assembly instructions
• Stack Buffer Overflow
Software Security
When is a program secure?

A. When it does what we want it to do
B. When we ensure that bad inputs do not result in unintended functionality
C. We need B + some more safeguards (what are some examples?)
D. We can never have a secure program
When is a program secure?

• Formal approach: When it does exactly what it should
  • not more
  • not less
• But how do we know what it is supposed to do?
When is a program secure?

• Formal approach: When it does exactly what it should
  • not more
  • not less

• *But how do we know what it is supposed to do?*
  • somebody tells us (do we trust them?)
  • we write the code ourselves (what fraction of s/w have you written?)
When is a program secure?

• Pragmatic approach: when it doesn’t do bad things

• Often easier to specify a list of “bad” things:
  • delete or corrupt important files (integrity)
  • crash my system (availability)
  • send my password over the internet (confidentiality)
  • send phishing email
When is a program secure?

• But .. what if the program doesn’t do bad things, but could?

• is it secure?
Weird machines

• complex systems contain unintended functionality

• attackers can trigger this unintended functionality
  • i.e. they are exploiting vulnerabilities
What is a software vulnerability?

• A bug in a program that allows an unprivileged user capabilities that should be denied to them.

• There are a lot of types of vulnerabilities
  • bugs that violate “control flow integrity”
    • why? lets attacker run code on your computer!

• Typically these involve violating assumptions of the programming language or its run-time
Exploiting vulnerabilities (the start)

• Dive into low level details of how exploits work
  • How can a remote attacker get a victim program to execute their code?

• Threat model: victim code is handling input that comes from across a security boundary
  • what are examples of this?

• Security policy: want to protect integrity of execution and confidentiality of data from being compromised by malicious and highly skilled users of our system.
Today: stack buffer overflows

• **Understand** how buffer overflow vulnerabilities can be exploited
• **Identify** buffer overflows and assess their impact
• **Avoid** introducing buffer overflow vulnerabilities
• Correctly **fix** buffer overflow vulnerabilities
Buffer Overflows

- An anomaly that occurs when a program writes/reads data beyond the boundary of a buffer
- Canonical software vulnerability
  - ubiquitous in system software
  - OSes, web servers, web browsers
- If your program crashes with memory faults, you probably have a buffer overflow vulnerability
Common Vulnerabilities and Exposures (CVE): security flaw that is publicly known

Critical Systems are written in C/C++
- OS kernels
- High-performance servers
  - Apache, MySQL
- Embedded Systems
  - IoT devices, “smart” vehicles, the MARs rover..

https://nvd.nist.gov/vuln/search
CS 31 Recap
Memory

• Abstraction goal: make every process think it has the same memory layout.
  • MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.

• Reality: there’s only so much memory to go around, and no two processes should use the same (physical) memory addresses.

OS (with help from hardware) will keep track of who’s using each memory region.
**Memory Terminology**

**Virtual (logical) Memory:** The abstract view of memory given to processes. Each process gets an independent view of the memory.

**Physical Memory:** The contents of the hardware (RAM) memory. Managed by OS. Only **ONE** of these for the entire machine!

**Address Space:** Range of addresses for a region of memory. The set of available storage locations.

**Virtual address space (VAS):** fixed size.

(Determined by amount of installed RAM.)
Memory

• Behaves like a big array of bytes, each with an address (bucket #).

• By convention, we divide it into regions.

• The region at the lowest addresses is usually reserved for the OS.
NULL: A special pointer value.

NULL is equivalent to pointing at memory address 0x0. This address is NEVER in a valid segment of your program’s memory.

- This guarantees a segfault if you try to dereference it.
- Generally a good ideal to initialize pointers to NULL.
What happens if we launch an attack where we load an instruction to execute at 0x0

A. Nothing will happen, this region is mapped to the NULL pointer, which does not have any effect
B. There will be some effect, but not necessarily devastating
C. This will have a devastating effect.
Memory - Text

• After the OS, we store the program’s code.

• Instructions generated by the compiler.
Memory – (Static) Data

• Next, there’s a fixed-size region for static data.

• This stores static variables that are known at compile time.
  • Global variables
Memory - Stack

- At high addresses, we keep the stack.
- This stores local (automatic) variables.
  - The kind we’ve been using in C so far.
  - e.g., int x;

Diagram:

```
0x0
Operating system
Code (aka. Text)
Data
X:        Stack
0xFFFFF
```
Memory - Stack

• The stack grows upwards towards lower addresses (negative direction).

• Example: Allocating array
  • int array[4];
Memory - Heap

• The heap stores dynamically allocated variables.

• When programs explicitly ask the OS for memory, it comes from the heap.
  • malloc() function
Instructions in Memory

0x0

Operating system

Text

Data

Heap

Stack

0xFFFFFFFF

funcA:
...
call funcB
...
funcB:
pushl %ebp
movl %esp, %ebp
...

Function B

Function A

...
Process memory layout

.text
  - Machine code of executable

.data
  - Global initialized variables

.bss
  - Below Stack Section
    - global uninitialized vars

heap
  - Dynamic variables

stack
  - Local variables
  - Function call data

Env
  - Environment variables
  - Program arguments

Operating system
  .text
  .data
  .bss

Heap

Stack
  Command line arguments
  Environment variables

0x0
0xFFFFFFFF
Process memory layout

.text
• Machine code of executable

.data
• Global initialized variables

.bss
• Below Stack Section
global uninitialized vars

heap
– Dynamic variables
stack
– Local variables
– Function call data

Env
– Environment variables
– Program arguments

```
int i = 0;
int main()
{
    char *ptr = malloc(sizeof(int));
    char buf[1024];
    int j;
    static int y; //similar to global vars
}
```
Process memory layout

.text
  • Machine code of executable

.data
  • Global initialized variables

.bss
  • Below Stack Section
    global uninitialized vars

heap
  – Dynamic variables

stack
  – Local variables
  – Function call data

Env
  – Environment variables
  – Program arguments

int i = 0;
int main()
{
  char *ptr = malloc(sizeof(int));
  char buf[1024]
  int j;
  static int y;
}

  • i -> data segment
  • ptr -> stack
    • data allocated on heap
  • buf -> stack
  • j -> stack
  • y -> bss
X86: The De Facto Standard

• Extremely popular for desktop computers

• Alternatives
  • ARM: popular on mobile
  • MIPS: very simple
  • Itanium: ahead of its time

• CISC
  • 100 distinct opcodes

• Register poor
  • 8 registers of 32 bits
  • only 6 general purpose
  • instructions are variable length
    • not aligned at 4 byte boundaries
    • lots of backward compatibilities
      • defined in late 70s
      • exploit code that no one pays attention to
  • we will use 32 bit because its more convenient.
Compilation Steps (.c to a.out)

C program (p1.c) → Compiler (gcc -m32 -S)

Assembly program (p1.s) → Assembler (gcc -c (or as = gcc’s assembler))

Object code (p1.o) → Linker (gcc (or ld))

Executable code (a.out) → Other object files (p2.o, p3.o, ...)

Library obj. code (libc.a)

You can see the results of intermediate compilation steps using different gcc flags.
Compilers

- Computers don't execute source code
  - Instead, they use machine code
- Compilers translate code from a higher level to a lower one
- In this context, C → assembly → machine code
# Object / Executable / Machine Code

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Machine Code (Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push %ebp</code></td>
<td>55</td>
</tr>
<tr>
<td><code>mov %esp, %ebp</code></td>
<td>89 E5</td>
</tr>
<tr>
<td><code>sub $16, %esp</code></td>
<td>83 EC 10</td>
</tr>
<tr>
<td><code>movl $10, -8(%ebp)</code></td>
<td>C7 45 F8 0A 00 00 00</td>
</tr>
<tr>
<td><code>movl $20, -4(%ebp)</code></td>
<td>C7 45 FC 14 00 00 00</td>
</tr>
<tr>
<td><code>movl -4(%ebp), $eax</code></td>
<td>8B 45 FC</td>
</tr>
<tr>
<td><code>addl $eax, -8(%ebp)</code></td>
<td>01 45 F8</td>
</tr>
<tr>
<td><code>movl -8(%ebp), %eax</code></td>
<td>B8 45 F8</td>
</tr>
<tr>
<td><code>leave</code></td>
<td>C9</td>
</tr>
</tbody>
</table>

Almost a 1-to-1 mapping to Machine Code
Hides some details like num bytes in instructions
Assembly
push %ebp
mov %esp, %ebp
sub $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
leave

int main() {
    int a = 10;
    int b = 20;

    a = a + b;

    return a;
}
Processor State in Registers

Information about currently executing program

• Temporary data
  ( %eax - %edi )

• Location of runtime stack
  ( %ebp, %esp )

• Location of current code control point
  ( %eip, ... )

• Status of recent tests %EFLAGS
  ( CF, ZF, SF, OF )

General purpose registers
- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

Current stack top
- %esp

Current stack frame
- %ebp

Program Counter (PC)
- %eip

Condition codes
- CF
- ZF
- SF
- OF
General purpose Registers

<table>
<thead>
<tr>
<th>Register name</th>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
<th>%eip</th>
<th>%EFLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits:</td>
<td></td>
<td></td>
<td>16</td>
<td>15</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td>%eax</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td>%edx</td>
<td>%dx</td>
</tr>
<tr>
<td></td>
<td>%bx</td>
<td>%si</td>
<td></td>
<td></td>
<td>%edi</td>
<td>%di</td>
<td></td>
<td></td>
<td>%esp</td>
<td>%sp</td>
</tr>
</tbody>
</table>

Six are for instruction operands

Can store 4 byte data or address value

The low-order 2 bytes \%ax is the low-order 16 bits of \%eax

Two low-order 1 bytes \%al is the low-order 8 bits of \%eax

May see their use in ops involving shorts or chars
Assembly Programmer’s View of State

CPU

32-bit Registers

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td>next instr addr (PC)</td>
</tr>
<tr>
<td>%EFLAGS</td>
<td>cond. codes</td>
</tr>
</tbody>
</table>

Registers:
- PC: Program counter (%eip)
- Condition codes (%EFLAGS)
- General Purpose (%eax - %ebp)

Memory

Addresses

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Data

Instructions

Program:
- data
- instrs
- stack

Memory:
- Byte addressable array
- Program code and data
- Execution stack
Stack Frame Contents

• What needs to be stored in a stack frame?
  • Alternatively: What must a function know?

• Local variables
• Previous stack frame base address
• Function arguments
• Return value
• Return address

• Saved registers
• Spilled temporaries
Stack Frame Contents

• What needs to be stored in a stack frame?
  • Alternatively: What must a function know?

• Local variables
• Previous stack frame base address
• Function arguments
• Return value
• Return address

• Saved registers
• Spilled temporaries
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Must adjust %esp, %ebp on call / return.
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  • pushl %ebp
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  • pushl %ebp
  • Set %ebp = %esp
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  • pushl %ebp
  • Set %ebp = %esp
  • Subtract N from %esp

Callee can now execute.
Frame Pointer

- Must maintain invariant:
  - The current function’s stack frame is always between the addresses stored in \%esp and \%ebp

- To return, reverse this:
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  • set %esp = %ebp
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  • set %esp = %ebp
  • popl %ebp
Frame Pointer

• Must maintain invariant:
  • The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  • set %esp = %ebp
  • popl %ebp

IA32 has another convenience instruction for this: leave

Back to where we started.
Frame Pointer: Function Call

Initial state

caller

pushl %ebp (store caller’s frame pointer)

callee

caller

caller’s %ebp value

movl %esp, %ebp (establish callee’s frame pointer)

subl $SIZE, %esp (allocate space for callee’s locals)
Frame Pointer: Function Return

Want to restore caller’s frame.

IA32 provides a convenience instruction that does all of this: `leave`
Functions and the Stack

Program Counter (%eip)

1. pushl %eip
2. jump funcB
3. (execute funcB)

Stack Memory Region

Function B

Stored eip in funcA

Function A

...

Text Memory Region

funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
...
call funcB
addl %eax, %ecx
...

funcB:
pushl %ebp
movl %esp, %ebp
...
movl $10, %eax
leave
ret
Functions and the Stack

1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip

Text Memory Region

```
funcA:
    addl $5, %ecx
    movl %ecx, -4(%ebp)
    ...  
    call funcB
    addl %eax, %ecx
    ...  
    funcB:
    pushl %ebp
    movl %esp, %ebp
    ...  
    movl $10, %eax
    leave
    ret
```
Functions and the Stack

6. (resume funcA)

Text Memory Region

```assembly
funcA:
   addl $5, %ecx
   movl %ecx, -4(%ebp)
   ... 
call funcB
   addl %eax, %ecx
   ... 
funcB:
   pushl %ebp
   movl %esp, %ebp
   ... 
   movl $10, %eax
   leave
   ret
```
Functions and the Stack

1. pushl %eip
2. jump funcB
3. (execute funcB)
4. restore stack
5. popl %eip
6. (resume funcA)

Text Memory Region

funcA:
addl $5, %ecx
movl %ecx, -4(%ebp)
...
call funcB
addl %eax, %ecx
...

funcB:
pushl %ebp
movl %esp, %ebp
...
movl $10, %eax
leave
ret
Functions and the Stack

1. pushl %eip  
2. jump funcB  
3. (execute funcB)  
4. restore stack  
5. poxl %eip  
6. (resume funcA)

Return address:
Address of the instruction we should jump back to when we finish (return from) the currently executing function.
Register Convention

• Caller-saved: %eax, %ecx, %edx
  • If the caller wants to preserve these registers, it must save them prior to calling callee
  • callee free to trash these, caller will restore if needed

• Callee-saved: %ebx, %esi, %edi
  • If the callee wants to use these registers, it must save them first, and restore them before returning
  • caller can assume these will be preserved
Putting it all together…

Callee’s frame.
- Callee’s local variables.
-Caller’s Frame Pointer
  - Return Address
  - First Argument to Callee
  - …
  - Final Argument to Callee
- Caller’s local variables.

Caller’s frame.
- …
- Older stack frames.

Callee Code
1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

Caller Code
1. save address of next instruction
2. push arguments
Implementing a function call

```
(main) (foo)

%eax 10

esp

main:
...  
sbl $8, %esp
movl $2, 4(%esp)
movl $1, (%esp)
call foo
addl $8, %esp
...  

foo:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl 8(%ebp), %eax
addl $9, %eax
leave
ret
```

Stack data

1
2
Stack
void main()
{
    bar("CS88");
}

void bar(char * in)
{
    char name[5]; // "CS88"
    strcpy(name, in);
}

bar:
    pushl %ebp
    movl %esp, %ebp
    subl $5, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    leal -5(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    leave
    ret
Data types / Endianness

x86 is a little-endian architecture

```
pushl %eax
```

```
%eax 0xfoo5ball
```

```
0x11 0xba 0x05 0x0f

1 1 1 1
```

Higher Memory Addresses
Buffer Overflows
Example 1

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {
    char nice[] = "is nice."
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```

What happens if we read a long name?

A. Nothing bad will happen
B. Something nonsensical will result
C. Something terrible will result
HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).

User Meg wants these 6 letters: POTATO. User Xada wants pages about "irl games". Unlocking secure records with master key 513098573363.

SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

User Meg wants these 4 letters: BIRD. There are currently 402 connections open. User Brendan uploaded the file "(" in car why". Note: Files for 11/11/38117.13 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD.
SERVER, ARE YOU STILL THERE?
IF SO, REPLY "NAT" (500 LETTERS).

User: Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038594". Isabel wants pages about snakes but not too long. User: Karen wants to change account password to "peanut."
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```

Load function arguments starting with the last argument
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball ;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa, 0xbbbbbrbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa, 0xbbbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaa,0xbbbbbbbb,argv[1]);
    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char **argv) {
    func(0xaaaaaaaa,0xbBBBBBBB,argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is “AAAAAAAAAAAAAAAAAA"

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa, 0xbbbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is “AAAAAAAAAAAAAAAAAA"

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa,0xbbbbbbbbb,argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAAA"

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaaa, 0xbbbbbbbbb, argv[1]);
    return 0;
}
```
Buffer Overflow example: If the first input is "AAAAAAAA\x95\x9b\x04\x08"

```c
#include <stdio.h>
#include <string.h>

void foo() { 0x08049b95
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbbbb, argv[1]);
    return 0;
}
```
Better Hijacking Control

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xffffffff,0xbbbbbbbb,argv[1]);
    return 0;
}
```
Better Hijacking Control

```c
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xfoo5ball;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaac,0xbbbbbbbb,argv[1]);
    return 0;
}
```

Jump to attacker supplied code where?
- put code in the string
- jump to start of the string
Shellcode

- Type of control flow hijack: taking control of the instruction pointer
- Small code fragment to which we transfer control
- Shellcode used to execute a shell
Shellcode

```c
int main(void) {
    char* name[1];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}
```

How do we transfer this to code?
Take the compiled assembly?
Payload is not always robust

Exact address of the shellcode start is not always easy to guess

Miss? Segfault

Fix? NOP Sled!
NOP Sled!

- NOP instruction: 0x90
- NOP sleds are used to pad out exploits
  - Composed of instruction sequences that don't affect proper execution of the attack
  - Classically the NOP instruction (0x90), but not restricted to that
- Why are the called sleds?
  - Execution *slides* down the NOPs into your payload
  - Overwritten return address can be less precise, so long as we land somewhere in the NOP sled
Small Buffers

Buffer can be too small to hold exploit Code

Store exploit code in:
• an environmental variable
• or another buffer allocated on the stack
• redirect return address accordingly
Putting it all together
Summary: Stack Code Injection

• Executable attack code is stored on stack, inside the buffer containing attacker’s string
  • Stack memory is supposed to contain only data, but...
  • For the basic stack-smashing attack, overflow portion of the buffer must contain correct address of attack code in the RET position
    • The value in the RET position must point to the beginning of attack assembly code in the buffer
      • Otherwise application will crash with segmentation violation
    • Attacker must correctly guess in which stack position his buffer will be when the function is called