Buffer overflow

Mohammad Mahzoun

University of Tehran

May 24, 2018

Introduction

What is a buffer overflow?

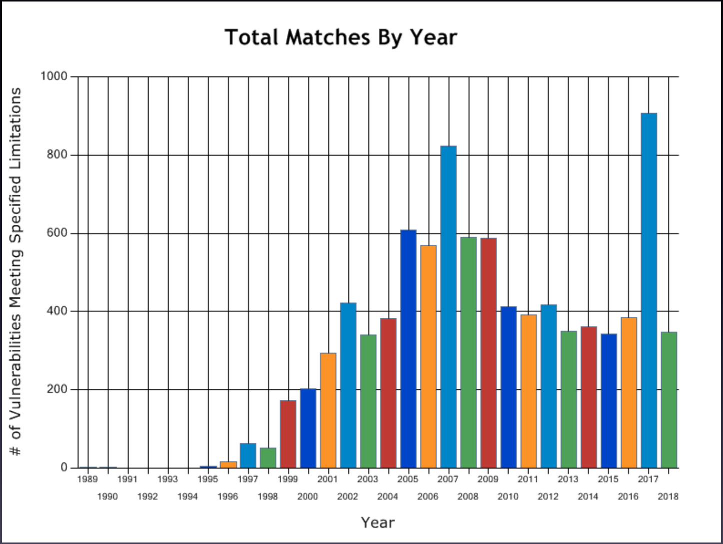
- A buffer overflow is a bug that affects low-level code, typically written in C and C++, with significant security implications.
- A program with this bug will simply crash.
- But an Attacker can do much worse!
    - **Steal** private information.
    - **Corrupt** valuable information.
    - **Run** arbitrary code.

History

History of buffer overflows

- Morris worm (1988)
  - Propagated across the machines using buffer overflow.
  - End result: $10-100M in damages
- CodeRed (2001)
  - Exploited an overflow in MS-IIS server
  - 300,000 machines infected in 14 hours
- X11 Vulnerability (2014)
  - The bug was in code for more than 20 years.

History



https://nvd.nist.gov/vuln/search/statistics

C memory layout

A typical memory representation of C program consists of following sections:

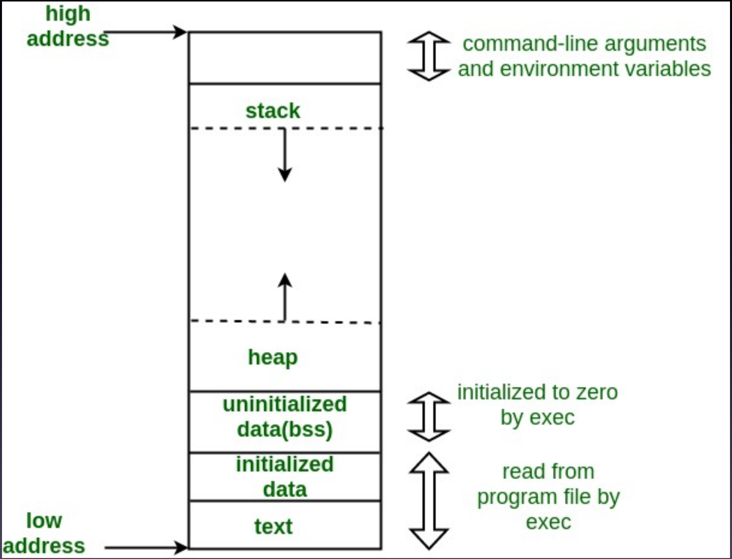
- Text segment
  - contains executable instructions.
  - Placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- Initialized data segment
  - virtual address space contains the global variables and static variables initialized.
- Uninitialized Data Segment
  - bss (block started by symbol)
  - all global variables and static variables that are initialized to zero or do not have explicit initialization

C memory layout

A typical memory representation of C program consists of following sections:

- Stack
  - local variables variables
  - saved information after function calls
- Heap
  - begins at the end of the BSS segment and grows to larger addresses from there.
  - managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size.

C memory layout

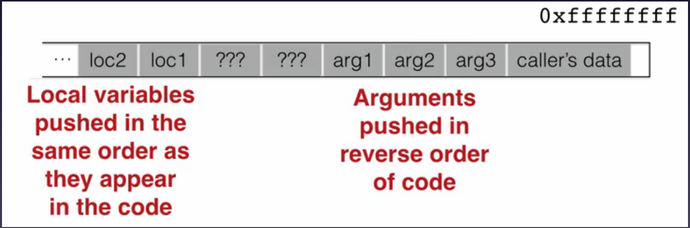

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Stack and function calls

- what happens when we call a function?
    - what data needs to be stored?
    - where does it go?
- what happens when we return from a function?
    - what data needs to be restored?
    - where does come from?

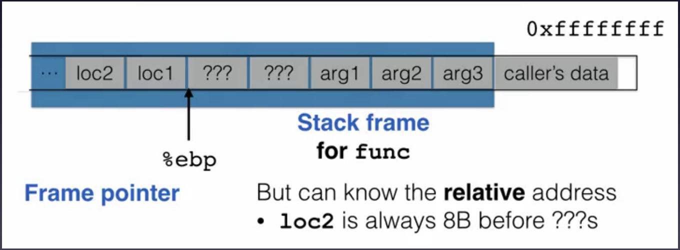
Stack and function calls

```
void func(char* arg1, int arg2, int arg3)
{
    char loc1[4];
    int loc2;
    ...
}
```


0xffffffff
...
loc2
loc1
???
???
arg1
arg2
arg3
caller's data
Local variables pushed in the same order as they appear in the code
Arguments pushed in reverse order of code

Accessing variables

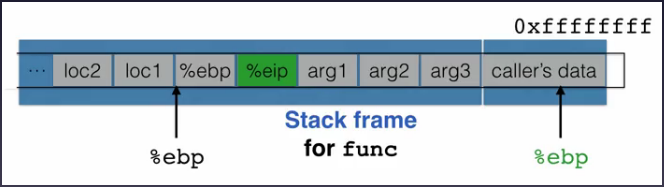
```
void func(char* arg1, int arg2, int arg3)
{
    ...
    loc2++; // Where is it? %ebp - 8
    ...
}
```



But can know the **relative** address
- `loc2` is always 8B before ???s

Returning from a function

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

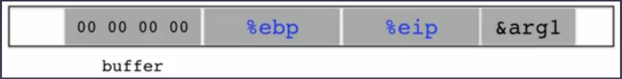


Buffer overflows

- Buffer
  - Contiguous memory associated with a variable or field
  - Common in C (Strings)
- Overflow
  - Put more into a buffer than it can hold
- Where does the overflowing data go?
- Well, now we know the memory layout ...

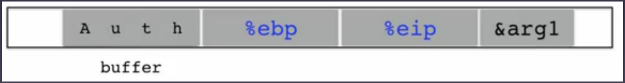
Buffer overflows

```
void func(char *arg1)
{
   char buffer[4];
   strcpy(buffer, arg1);
   ...
}
int main()
{
    const *mystr = "Authme!";
    func(mystr);
     ...
}
```



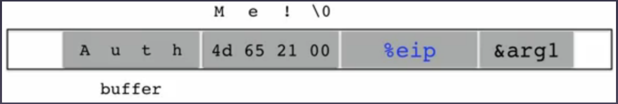
Buffer overflows

```
void func(char *arg1)
{
   char buffer[4];
   strcpy(buffer, arg1);
   ...
}
int main()
{
    const *mystr = "Authme!";
    func(mystr);
     ...
}
```



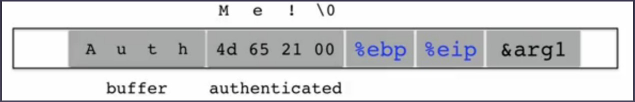
Buffer overflows

```
void func(char *arg1)
{
   char buffer[4];
   strcpy(buffer, arg1);
    ...
}
int main()
{
     const *mystr = "Authme!";
     func(mystr);
      ...
}
```



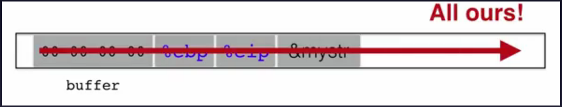
Buffer overflows

```
void func(char *arg1)
{
   int authenticated = 0
   char buffer[4];
   strcpy(buffer, arg1);
   if(authenticated) { ...
}
int main()
{
    const *mystr = "Authme!";
    func(mystr);
     ...
}
```



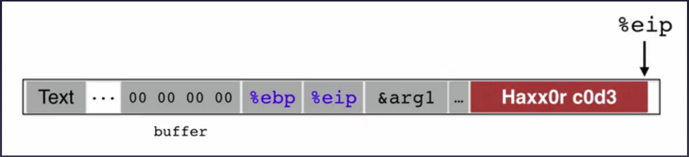
Even Worse!

Attacker can inject his code and arrange for the program to execute it!



Code Injection

```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



Inject Code

Loading Code:

- Machine code
- Can’t contain all-zero bytes
- Can’t use loader

What should we inject?

- general-purpose shell!

Code Injection

```
#include <stdio.h>
int main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execv(name[0], name, NULL);
}
```

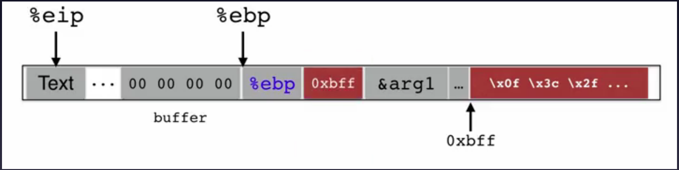
Code Injection

```
00000000    31C0          xor eax,eax
00000002    50            push eax
00000003    682F2F7368    push dword 0x68732f2f
00000008    682F62696E    push dword 0x6e69622f
0000000D    89E3          mov ebx,esp
0000000F    50            push eax
00000010    53            push ebx
00000011    89E1          mov ecx,esp
00000013    99            cdq
00000014    B00B          mov al,0xb
00000016    CD80          int 0x80
```

Inject Code

Running Code:

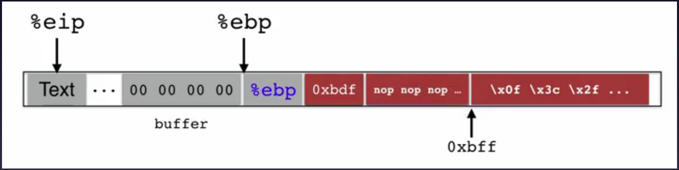
- We can not use: jump into my code instruction
- We do not know where exactly our code is?



Inject Code

Find our code address:

- Guess it?!
- use *nop*



Questions

Questions?