Co-funded by the Erasmus+ Programme of the European Union

ISSES – Information Security Services Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

MEMORY ATTACK FUNDAMENTALS

COMPUTER SECURITY

Lecture 9

ISSES 2017-2020
Novi Sad, Serbia

Sources

- Part of these lecture notes has been inspired by AlephOne's Smashing the Stack for Fun and Profit.
- Part of the examples and structe of these lecture notes has been inspired by the lecture notes made pubic by Dr Avinash Kak and used at his course at Purdue University.
- The http://www.scs.stanford.edu/brop/bittau-brop.pdf, and the https://download.vusec.net/papers/blindside_ccs20.pdf papers are used as sources.
- The UMBC 2018 Spring CTF competition and its solution has been used as a source of or inspiration for examples.

Information Security Services Education in Serbia (ISSES)

9.1 BUFFER-BASED ATTACKS

The importance of buffer overflow attacks

- Nearly every prominent security flaw rests on a buffer-based attack, usually an overflow.
- It makes sense: to attack a system if the rules it is coded by are functioning correctly (and we cannot break the system by finding a loophole in its security rules) we have to break the system by somehow breaking out of the space given to us for our data and into the system's internal state.
- The main (but by no means only) way to do this is to overflow a buffer, or rather, to write past the end of some data structure and into memory that belongs to the internal part of the system that's not meant to be user-facing.
- Then we can influence parts of memory that are not ours and, by doing that, cause great harm to the correct operation of the system.

How come these attacks exist?

- So why not check for boundaries? In Java we can try to do this:

```
public class HelloWorld{

    public static void main(String []args){
        String[] tst = new String[5];
        tst[7] = "Experiment.";
    }
}
```

- And get a java.lang.ArrayIndexOutOfBoundException for our troubles.
- Why can't all access be like that?
- Performance and architecture

Performance and architecture

- Each check slows down access dramatically: it's not only more things to do but it also introduces a branch, and there's nothing a fast, modern CPU hates more than a branch.
- An fundamentally, if our code gets converted into machine code, for machine code, accessing any memory location in the address space of our process is fundamentally equivalent.
- The only way that we can prevent this is if we interpose something between the machine and our code, which is fine, but if we do that, we lose major performance boosts.
- That said: more secure programming langauges do exist, it's just that most system code is written in or relies on C/C++ code which... isn't secure. At all.

Overflow and underflow

- Mostly we'll talk about overflows: we write past the end of a buffer, thus clobbering memory that should be something else, and by doing that we cause effects that should not be possible.
- Underflow is similarly hazardous and comes from having two different bits of our program's code treat the same memory differently.
- A canonical example is allocating 2n bytes as a string buffer, putting n bytes worth of string in it, and then in another part of the code we retrieve and process all 2n bytes.
- This means we have an unpredictable (and possibly manipulable) n bytes trailing our actual buffer which may reveal secret information or be a malicious data injection vector.

Overflow example: MS17-010

- This is a classic example of a buffer overflow vulnerability.
- It's Microsoft this time around, but generally there is no platform spared trouble with buffer overflows.
- Security researchers have managed to disassemble the affected code (see also https://www.exploit-db.com/exploits/41987) which permits a specially crafted message sent to a SMBv1 server to execute arbitrary code.
- The trick is that the function accepts user input, makes unwarranted assumptions about it (and where it will fit in memory) and then copies it.
- This can be exploited and, indeed, was by WannaCry ransomware to such an extent that it made the news.

Where's the vulnerability

```
unsigned int __fastcall f(int a1, int a2)
{
  int v4; // edi@1
  _BYTE *v5; // edi@1
  unsigned int result; // eax@1

  v4 = a1 + 8;
  *(_BYTE *)(a1 + 4) = *(_BYTE *)a2;
  *(_BYTE *)(a1 + 5) = *(_BYTE *)(a2 + 1);
  *(_WORD *)(a1 + 6) = *(_WORD *)(a2 + 2);
  _memmove((void *)(a1 + 8), (const void *)(a2 + 4), *(_BYTE *)(a2 + 1));
  v5 = (_BYTE *)(*(_BYTE *)(a1 + 5) + v4);
  *v5++ = 0;
  _memmove(v5, (const void *)(a2 + 5 + *(_BYTE *)(a1 + 5)), *(_WORD *)(a1 + 6));
  result = (unsigned int)&v5[*(_WORD *)(a1 + 6) + 3] & 0xFFFFFFFC;
  *(_DWORD *)a1 = result - a1;
  return result;
}
```

The stack and the heap

- As you are doubtless aware, memory you allocate can be on the stack or in the *heap.*
- (Those among you who are heavily into more exotic uses of C++, know that custom allocators may cause this neat situation to get considerably more complex, but that's not relevant for us tight now)
- The stack is where local variables, function arguments, plus all the other niceties needed for a stack frame and function calls to exist live.
- The heap is where memory is allocated dynamically through calls to malloc/new and relatives.
- Both areas of memory are prone to overflow conditions.

Comparing overflows

- While both overflows are dangerous, it's the stack buffer overflow that's most commonly exploited.
- Why? Because, as we've already covered when we first smashed a stack, the stack is where the fundamental security flaw of the Von Neumann architecture most readily manifests: it's a place we store both data (including outside-world data) and code, in the sense of using it to store a return address.
- This is not to say that the heap buffer overflow is harmless, to the contrary, it causes issues like CVE-2015-1271 which results from a heap buffer overflow in the pdfium library, allowing a maliciously malformed PDF faile to trigger the vulnerability.

Can you spot the vulnerability?

```
void CFX_BinaryBuf::ExpandBuf(FX_STRSIZE add_size)
{
    FX_STRSIZE new_size = add_size + m_DataSize;
    if (m_AllocSize >= new_size) {
        return;
    }
    int alloc_step;
    if (m_AllocStep == 0) {
        alloc_step = m_AllocSize / 4;
        if (alloc_step < 128 ) {
            alloc_step = 128;
        }
    } else {
        alloc_step = m_AllocStep;
    }
    new_size = (new_size + alloc_step - 1) / alloc_step * alloc_step;
    FX_LPBYTE pNewBuffer = m_pBuffer;
    if (pNewBuffer) {
        pNewBuffer = FX_Realloc(FX_BYTE, m_pBuffer, new_size);
    } else {
        pNewBuffer = FX_Alloc(FX_BYTE, new_size);
    }
    if (pNewBuffer) {
        m_pBuffer = pNewBuffer;
        m_AllocSize = new_size;
    }
}
```

Hints: What happens if the call to FX_Realloc should fail?

What do allocation functions generally return when they can't allocate more memory?

What's writing *anything* in a empty buffer?

Limited exploitability

- Heap buffer overflows are harder to exploit because while you can clobber a return address on a stack, the heap doesn't have anything like that.
- The vulnerability we just saw is a DOS vulnerability, meaning someone can create an input that crashes our program which is *annoying* but not actively dangerous.
- However, a heap buffer overflow can be used to manipulate the structure *around* an allocated chunk of memory (which contains status bits and the size of an allocated chunk) which can force incorrect behavior out of, say, free.
- This, in turn, can give us the ability to alter areas of memory we really should not have access to, including the stack by a roundabout way or the procedure linkage table.

Refresher: how a stack works

- Each thread in a process gets its own stack.
- There's nothing *magic* about stacks. It's just an area of memory which is treated in a special way.
- How? The address to the stack is kept in a special register and this value is used and modified in stack manipulation instructions like, say, 'pushl' which is a way to push a 32-bit word onto the stack.
- If we imagine code that looks like this:

```
int f(int a, int b, int c){
    int x = 100;
    return a + b + c;
}

int main(){
    f(1, 2, 3);
    return 0;
}
```

Refresher: how a stack works

- How does the stack look like at the point after I've asigned 100 to X? The stack contains, in order:
  - 100 [This is the value of x, and this is what %esp would point to in a 32-bit program. We are talking 32-bit here because it is easier to explain.]
  - EBP [this is the old value of the basis pointer, it tells the calling function where to return to so as to be able to cleanly access its local variables]
  - Return_addres [this is the value of the instruction pointer register at the time of function call: it's where control will return to once 'ret' has been executed]
  - A, B, C [the values for the arguments]
  - Main_return_address [the return address of the main function]
- This is just a refresher: *you should know all of this.*

Refresher: how a stack works

- To generalize, the stack consists of a number of *frames* each of which describes the state of a function in the process of being exectuted (here main and f) and each of which but the last (just main for us) is interrupted by a call to a subsequent function.
- The general structure is consistent, at the top of the stack are local variables in order of use in the enclosing scope, then there is the saved EBP, then a return address, then arguments.
- This is the fundamental structure we exploit, and the structure compiling creates.

Assembly for the sample C code

```
_Z1fiii:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    subl    $16, %esp
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    $100, -4(%ebp)
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    addl    %eax, %edx
    movl    16(%ebp), %eax
    addl    %edx, %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
```

```
main:
.LFB1:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    pushl   $3
    pushl   $2
    pushl   $1
    call    _Z1fiii
    addl    $12, %esp
    movl    $0, %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
```

Note that this is not the complete code (some infrastructure has been left out). Likewise, this code is not hand-written, it is what GCC generates. Finally, certain automated protections have been purposfully disabled for the purposes of this code. The command line for generating it is: gcc -O0 -m32 -S -o test1.S test1.cpp -fno-stack-protector -fcf-protection=none, you will understand what they mean next class.

Plan for mounting an attack

- First we need a vulnerability. How to find one? If we have source code we can hunt for anywhere where buffers are used with implicit unsafe assumptions.
- Say we allow the user to enter whatever they like and store it in a finite-size buffer. This is a vulnerability.
- Certain C functions are, by their very nature, a vulnerability:

gets	strcpy
strcat	sprintf
scanf	fscanf
vfscanf	vsprintf
vscanf	vsscanf
streadd	strecpy
Note: Not all are equally vulnerable	

Plan for mounting an attack

- In case you do not have the source code, you will have to run the program with malformed inputs of various sizes seeing what it takes to crash the program.
- If you find a crash, you have found a potential weakness you can exploit.
- Your next step is to attach a debugger to running code and carefully see what happens to the stack and the various register values at the point of the crash.
- It helps if your malformed data has a unique pattern which you can instantly detect if it should overwrite anything crucial.
- Once you've determined how much filler you need to start affecting register values (if that ever happens) you can start exploiting the bug.

Plan for mounting an attack

- Once you have found your vulneralibility you need to study how it works and be confident you can use it to write arbitrary values to your stack. Once that is done, your next step is to carefully analyze the code as it runs and determine the appropriate offsets to your code so that your return address can be altered apropriately.
- Once you have that you can send the execution path into memory you control.
- This constitutes a break of security but your *payload* isn't ready yet.
- You can tell the system to execute any code you want, but what is it you want in the first place?

Plan for mounting an attack

- What you want, usually, is control of the system, or at least some of it to start with.
- This usually means gaining remote access to a given system which it is not inclined to give to you.
- Thus, the most common payload for this sort of attack is *shellcode,* a binary string that contains the opcodes of code which executes the bourne shell (pretty much guaranteed to be present).
- Once shell has been executed, you have access to the system at the level of the owning user of the process you just hacked.
- From here you either have what you came for, or you can run some sort of privilege escalation attack.

Manufacturing shellcode

- To step back for a second, the most likely source of shellcode is online. A 27-byte 64-bit shellcode is "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05".
- However, for education purposes, let's figure it out, step by step.
- Step one is to figure out what the system does when we run a shell.
- To this end, we can compile a simple c program that runs the execve system call to spawn a shell process.
- The program is trivial:

Execve and shells

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

Note the somewhat odd use of str: this is because we need to supply an argv array which is null terminated and the first element of which is always how the piece of software in question has been invoked.

```
EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *pathname, char *const argv[],
                  char *const envp[]);
```

Getting to assembly

- We don't want C code.
- We don't even want C code that's been compiled regularly for this.
- We want to minimize the assembly in question as much as possible, therefore we need to know precisely what the system does.
- Luckily, this is not difficult: the secret is to compile the code using the –ggdb and –static options. This makes the code gdb friendly and the static option incorporates all the library code into our executable.
- Why? So we can peek into the implementation of excve and see how the system call is being made *exactly.*
- We can do this by using the disas comand in gdb

Manufacturing shellcode

- Once we have access to what exceve does, the next step is to manufacture a self-contained bit of assembly which, when executed, does the same thing. This is not as easy as it seems.
- The chief difficulty isn't issuing the correct commands (there's no real skill in loading '11' into the A register and calling int 0x80 or the equivalent on more modern systems)
- The difficulty is passing the string into the system call since that requires us to know the address which we, by definition do not have.
- To solve this problem without guesswork which is almost certain to fail, the method that can be used is exploiting the call instruction.

Aleph One's approach

```
jmp    0x26
popl   %esi
movl   %esi,0x8(%esi)
movb   $0x0,0x7(%esi)
movl   $0x0,0xc(%esi)
movl   $0xb,%eax
movl   %esi,%ebx
leal   0x8(%esi),%ecx
leal   0xc(%esi),%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
call   -0x2b
.string \"/bin/sh\"
```

Note that this approach is obsolete in its details, but not in its main idea.

What we do here is immediatelly use a relative jump to jump 0x26 locations ahead to the call instruction. This, in turn takes not an absolute but a relative address (it's allowed to) and jumps back 0x2b places, but not before pushing onto the stack its return value the address of the string, /bin/sh.
Once that's done, we have the address, all that's left is to null-terminate the string, place the long word of all nulls to play the part of the NULL value we need to send.
The new just load the appropriate syscall index into %eax (11 for execve) and then the relevant addresses into the other registers and initiate a syscall which, in this instance, is done using the old-style syntax you are probably familiar with from your first-year courses. If execve succeeds, this process is replaced with a new one, if not we reach 'exit' and quit cleanly.

Patrick Shaller's approach

```
xor  %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov  %esp, %ebx
push %eax
push %ebx
mov  %esp, %ecx
mov  %eax, %edx
movb $0xb, %al
int  $0x80
```

This is a less elaborate but more ingenious solution. It creates a 0 on the stack without having to mention 0 (we'll see why later) by xor-ing two values together.
It then pushes onto the stack two confusing-seeming hexadecimal numbers: this is just the /bin/sh string in little-endian order. In other words, instead of having the string, the shellcode *creates* the string and, as a bonus, knows where it is: on the stack.
It then places its address in in the B-register, modifies the stack to have another zero, then the address to the string. Finally the pointer to the pointer to the string is placed in C register and the zero we had in eax into the D-register. This gives us a full list of execve parameters, so we put the syscall index where we ought to, and we initiate the system call.

Manufacturing shellcode

- Now that we have the assembly for our shellcode, we need to compile it and then use the objdump –d command to get access to the opcodes of our program.
- Once we have the opcodes it's a trivial task to turn them into individually-tailored bytes for our string using the \x escape character. AlephOne uses the following as his shellcode, therefore:

```
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
                   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
                   "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Manufacturing shellcode

- The last step is to verify that your shellcode does not contain any forbidden characters.
- Forbidden characters?
- Simple: to actually exploit shellcode we will have to feed it into the process we are attacking somehow.
- Chances are, we will be doing this through an input of some description.
- This means that, at a minimum, we need to avoid all NULs since those terminate strings.
- And, depending on how the data is being processed, we may have to avoid other characters as well, such as newlines, spaces, and so on.
- Provided we can avoid any foribiden characters, our payload is ready.

Injecting payload

- We only have one thing left to do: guess.
- The return address we need to clobber onto our stack is unknown. It's somewhat predictable, but not guranteed to be guessable.
- Therefore, we will construct an input to our vulnerable piece of code consisting of NOP instructions (landing on those is harmless) followed by our shellcode, followed by the rest of the buffer filled with return adresses.
- We still have to guess those, but as long as it lands somewhere in the nop code, we are safe.
- Careful analysis may help us guess with more certainty, but in general repeated attempts are something this sort of attack requires.

Information Security Services Education in Serbia (ISSES)

9.2 Advanced memory attacks: RTL, ROP, and printf-based

Why do we need anything else?

- If we identify a buffer overflow, aren't we already in control?
- I mean, we may need a privilege escalation attack, perhaps, but aside from that, a buffer overflow is all we need to inject shellcode and gain access.
- Why the need for other attacks?
- The next lecture will go into extended detail on this topic, but let's just say that OS authors don't sit idle while people mount increasingly elaborate stack-smashing attacks.
- A number of automated defenses can stop our injection and chief among them is blocking the executability of the system stack memory area.

Non-executable bits

- Simply put, the memory page where the system stack area is, is marked to be non-executable.
- This is supproted at the hardware level and, as a result, we can't bypass it through the OS
- If the relevant page is so marked we can't treat that region as code no matter what.
- This defats the simple exploit, since we've relied on our ability to construct code on the stack and then to just hack the return address so it points back at it.
- Given that the system will just crash if we try to execute code in a protected area, constructing shellcode won't help.
- What *can* we do?

Return-to-libc

- One solution is charmingly direct: okay, so the stack may be safe, but we can always jump somewhere else.
- Where? Well, what happens if we change the return address to something that is guaranteed to be predictable and executable.
- A wonderful example of that is libc: the standard C library that's always there, always available, and highly predictable.
- We still have access to the stack meaning that not only can we engineer the return address to suit us, we can also engineer the parameters.
- This means that, with a little bit of care, we should be able to call an arbitrary libc function with arbitrary parameters.

Return-to-libc

- How can we use that?

```
NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);
```

All we need is the address of the system command and how to set up the stack. We could disasemble this code, but since it operates through the PLT (which we explained previously) that represetns a dificulty. Luckily, gdb can help us here.

It is possible to get the addresses of everything we need with relative ease: both the system command and even the /bin/sh string.

The trick is gdb's ability to read values of just about anything our process has access to, and to search memory.

Finding addresses

- In newer versions of gdb it is possible to search through memory.
- But even in older versions it's only a question of dumping a region of memory to a file, searching through it, and finding the offset with the necessary data.

```
(gdb) break main
Breakpoint 1 at 0x80483c0: file test.c, line 4.
(gdb) run
Starting program: /home/veljko/test

Breakpoint 1, main () at test.c:4
4           system("sh");
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7dfb6e0 <system>
(gdb) dump binary memory tst.bin 0xb7dfb6e0 0xb7f00000
```

Other ways of loading in /bin/sh

- It's possible to sneak in the requisite string through a system variable.
- You can get the address of a system variable through a trivial piece of code:

```
int main(){
    char* tv = getenv("TESTVAR");
    if (tv)
        printf("%x\n", (unsigned int)tv);
}
```

This will print an address which is stable between invocations, meaning that it will always be the same, no matter what.

Return oriented programming

- ROPs are a variation on the return-to-libc attack that, instead of trying to use existing libc code go back to constructing shellcode (or any-other-purpose code).
- Except, instead of doing so on the stack (since that's likely not to be executable) you search the program's entire memory image, specifically, all the executable parts of it, and you look for something that fits what you need.
- After all, nothing stops you from jumping into the middle of code rather than its start.
- These usable parts are called 'gadgets' and chaining them together can, with some ingenuity, cause the attacked program to do just about anything you want it to.

Return oriented programming

- What makes a good gadget?
- The key part is that it does something we can use and that it ends with a ret instruction.
- That part is crucial, as it lets us go back to whatever address we have pushed on the stack.
- This means that, with carefully managing we can chain calls to ROP gadgets in near-arbitrary order, as long as we have control over the contents of the stack.
- Doing so manually is of course quite difficult.
- This is why ROP exploits are not constructed manually but through the use of handy tools. One such tool is Jonathan Salwan's ROPGadget
- https://github.com/JonathanSalwan/ROPgadget

Expert tools

- A combination of RTL and ROP exploitation can be used to identify an exploitable gadget for *any* glibc platform you might care for.
- Instead of doing it yourself, you can use the one_gadget tool made by github user 'david942j'
- https://github.com/david942j/one_gadget
- This ruby program takes any given glibc implementation and finds addresses that drop us into shell.
- It's also capable of telling us what contraints are needed for this to work so that we can set up things accordingly.

OneGadget

```
$ one_gadget /lib/x86_64-linux-gnu/libc.so.6
# 0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
# constraints:
#   rsp & 0xf == 0
#   rcx == NULL
#
# 0x4f322 execve("/bin/sh", rsp+0x40, environ)
# constraints:
#   [rsp+0x40] == NULL
#
# 0x10a38c execve("/bin/sh", rsp+0x70, environ)
# constraints:
#   [rsp+0x70] == NULL
```

Spoiling our party

- Detailed analysis of various defense mechanisms is left for later, but for now, let us focus on the crux of the problem: There exists a technology known as ASLR (address space layout randomization) that makes it so we can't guess the address of anything.
- If we can't guess the address of anything we can't really do *any* of the attacks we've been talking about.
- They all rely on us knowing the layout of, say, libc or some other linked library in memory.
- No such knowledge – no exploit.
- This is why ASLR is part of most modern operating systems.
- Does this mean all of this is obsolete?

Exploiting memory leaks

- In a word, no.
- The simplest way to work around this is to persuade the vulnerable program to send us the address of some function, any function in, say, libc.
- Addresses are randomized, certainly, but only at the level of a base address.
- This means that the place in memory where libc is is shifted by some, unpredictable number (you will see code that implements this next lecture), but within libc, wherever it is, the internal layout remains the same.
- Then, if we have the address of anything we can compute the base address of libc, and from that, the address of any gadget or function we might want.

How to leak memory?

- If we control the instruction pointer through the stack, we can use it to call a, say, puts function.
- We then give that puts function, as a parameter, the GOT address of the puts function itself (why? Because if we are calling puts than it's bound to be in the GOT)
- If we can get input back we can decode it (by simply padding from 6 bytes of a canonical address to 8 bytes of a full u64) and get the puts address.
- Since we have control over our program, we can just cause main to start going again (thus preventing the program from terminating just as it gave us something interesting) and use the calculated glibc address to construct a new payload that produces the desired effect.

GOT?

- To understand GOT and PLT you need to understand modern linkers.
- To run a dynamically linked progarm the way a modern compiled bit of C code works is to have two tables, the PLT and the GOT.
- Both the PLT and the GOT are fixed in memory (because they have to be) and have a predictable structure with each index in the respective table corresponding to a function.
- A given PLT entry (procedure linkage table) consists to a jump to the addres contained in the GOT (global offset table) at the same index, then code to prepare arguments for a resolver function, then a jump to the special PLT[0] location.
- PLT[0] just calls the resolver.

Resolver

- The resolver is the code responsible for talking with the OS and getting the actual address of the function.
- Once called it will change the code in the GOT
- Normally the GOT address is just the address of the 'prepare resolver' step in the relevant PLT entry
- Once the resolver is called, this is changed to the address of the actual function. This is the precious data we want.
- This means that the first call to PLT loads the function address, and each subseqnet call just calls the function instantly.

Beyond memory leaks

- This is above our level of understanding, but it is possible to use ROPs even in instances where we don't have a direct leak at all.
- The first technique is the BROP attack (Blind ROP) detailed here: http://www.scs.stanford.edu/brop/bittau-brop.pdf
- This paper describes a technique (implemented by the authors in a tool called Braile: http://www.scs.stanford.edu/brop/braille.rb) which requires only that a server have a stack overflow error, that we know a string which crashes it, and that it is a server which is automatically restarted once it crashes.
- That's all it needs to break through, even in the presence of ASLR, non-executable stacks, and stack canaries (next class).

Further beyond memory leaks

- It's possible to use a similar probing technique to attack even targets that are crash-sensitive, i.e those that do not restart on their own after a crash.
- How? Leaks using a specter-like vulnerability (we talked about it some lectures ago).
- The details are complex past the bounds of this course, and the field is the subject of active research.
- Details are available at: https://download.vusec.net/papers/blindside_ccs20.pdf see also https://www.vusec.net/projects/blindside/ for videos
- The blindside tool itself is available at https://github.com/vusec/blindside

A final obstacle

- All of this relies entirely on being able to, once we have the address, jump in the middle of some useful bit of code and then jump back out.
- This despite the fact that the place we are jumping to isn't *intended* as a jump target.
- After all, any piece of code written in C or C++ should have a failry clear idea of where it can jump: starts of functions, at the site of loops or branches and at the site of labels, should that author have taken leave of their senses and used a goto.
- This means we can differentiate legitimate branch targets from illegitimate ones and prevent any jump that we don't expect (more on this next class).
- Are we done *now?*

Nearly

- Intel's CET, which is a novel security technology is incredibly powerful.
- It's not universal quite yet, but it promises to defend from both jumping to arbitrary addresses and from calling arbitrary addresses.
- It does this through maintaining a whole different stack (a shadow stack) for return addresses which is not amenable to alteration.
- The endbranch instruction, on the other hand prevents us from jumping to places that aren't valid branches.
- With universal application (a tall order!) CET can stop us from leveraging the usual sort of attack completely, provided the software support is implemented without bugs (another tall order).

The future

- CET support is far from general and no general attack for it is known.
- It's certainly going to make looking for ways to defeat it the main occupation for security researchers in years to come
- However, *if* it is fully implemented and supported in its entirety with code that can be relied upon, it will make ROP/RTL type attacks obsolete.
- This will shift focus on, among other things, attacking the program's data, attempting to shift execution to the path we want by manipulating what the program sees.

Printf attacks

- A brief aside from cutting edge memory exploitation and the future of hardware protection into something much more familiar: the humble prtintf.
- Probably the first function you ever called with an entirely harmless purpose.
- Despite that, printf can be used to mount attacks
- Can you spot the vulnerability?

```c
#include <stdio.h>

int main(){
    char buff[80] = {0};
    scanf("%s",buff);//Buffer overflow, too
    printf(buff);
    return 0;
}
```

What's the problem?

- The compiler and the runtime do *no checking whatsoever* for a mismatch between the format string and the parameters actually given to printf.
- The compiler may warn about it, certainly, but the runtime has no way of knowing what we are up to.
- As a result, we can do this:

```
OS/prg/pred
▶  ./test
%08x%08x%08x%08x%08x%08x%08x%08x
0000000a000000000000000000000000466f32a078383025783830257838302578383025⏎
```

The input we provide is in red. What is the output? Depends. Either the stack or the first general-purpose registers followed by the stack. Why? Printf's variadic. It just takes as many parameters as it needs from wherever parameters for functions are kept: registers and stack (for 64-bit) or just the stack (for 32-bit).

Arbitrary memory read

- We can read arbitrary memory addresses too
- We just send into the string printf will process a string whose ASCII values consist of the address we wish to read.
- Then we execute the printing of as many %x as there is offset between the start of the input string and the end of the address.
- Then we print a %s. This will dereference whatever it pops from the stack which just happens to be the buffer address we constructed and start reading, byte by byte, until we get to a null character.
- With some patience and skill you can use this to leak memory from a process if you are lucky enough to discover a printf vulnerability.

Arbitrary memory write

- It's not just reading. With some effort it is possible to modify memory to while not quite *arbitrary* value, certainly a range of values.
- The %n format paramter is used to write in its place not a value of a variable but the number of bytes printed hitherto to the address provided on the stack.
- If we use the trick we used in the previous example to align bytes on the stack according to our needs, then we can use %n to fill that location with the number of bytes we have printed up to that point.
- By manipulating the contents of the format string we can place any of a range of values into the given address. This isn't an arbitrary value (unless we have a ~4GB input buffer to play with, an unlikely contingency), but there is choice.

Milking the format string

- Do we really need 4GB?
- Not *quite.* Format with specifications give us the ability to cheat a little bit.
- Well. More than a little.

```
printf("%16384.2d\n",5);
```

Yes, that's a 16384-character padding in front of our 5.
This means we can turn anything we can print into a near-arbitrary number of characters.
This allows us to turn the %n exploit into arbitrary value editing and can be used as a method to smash the stack.
The scenario may seem unlikely, but in truth printf-like values are often used to specify templates in various tools. If this information goes straight into a printf (including printf that's not called directly but through a wrapper from some other programming language) that can be used to manipulate memory and achieve an exploit.