I thought programs were supposed to be more secure now? Is it really even practical to try to do exploit development?

* Yes if you are skilled. Stat analyzers don’t catch everything. Also, many people don’t use them, do security testing on their products, or know anything about secure coding. So there tends to be several vulnerabilities in all software written.
* What about programs that utilize security measures like DEP, ASRL, and SEH?

Do we need debuggers to develop exploits?

* Yes. Even if we know the buffer is say 1000 bytes long, we still don’t know how we could potentially execute our own code. We need to put it in a debugger to know where to jump around, what parts of execution we can change, etc etc.

What is the difference between a disassembler, debugger, and a decompiler?

* [**Disassembler**](https://reverseengineering.stackexchange.com/questions/tagged/disassemblers):

A disassembler is a software tool which transforms machine code into a human readable mnemonic representation called assembly language.

* [**Debugger**](https://reverseengineering.stackexchange.com/questions/tagged/debuggers):

Debuggers allow the user to view and change the running state of a program.

* [**Decompiler**](https://reverseengineering.stackexchange.com/questions/tagged/decompiler):

Software used to revert the process of compilation. Decompiler takes a binary program file as input and output the same program expressed in a structured higher-level language.

So what tools can we use to get these things done?

* There are certain tools that can do all 3 of these things. IDA Pro and x64dbg are the two most popular at the moment. We will be using x64dbg because IDA Pro is extremely expensive and neither one is incredibly superior to each other. Immunity debugger and evans debugger also seem like great tools.
* “Detect it easy” tells us the compiler and linker so we know what to put in x64dbg

What tools do we need to use to exploit our target systems?

* For simplicity, all of our exploits will be written in python. You can write them in c/c++ if you like.

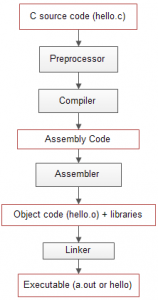
So we need to know two main things before getting started

* How are programs built/run?
* How does the operating system do all of these things with assembly instructions?

So are there any good practice or tutorials?

* Book: Practical reverse Engineering
* Modern Windows Exploit Development
* Good primer for c/assembly/exploitdev: <http://www.pwnthebox.net/> & <https://www.youtube.com/watch?v=75gBFiFtAb8> & <https://www.corelan.be/index.php/articles/>
* Tutorials and challenges: <https://tuts4you.com/e107_plugins/download/download.php?list.17>

# How do programs get built/run?



When going from source -> executable 4 files are created

* .c – source code
* .i – intermediate. Does the includes and defines. Input into compiler.

COMPILER START

* .s – source. assembly language. Compiler output. Optimized and removed dead code. Strips variable names.

ASSEMBLER START

* .o –object file. What assembler produces. Just pure 1s and 0s

LINKER START

* .exe – what the linker generates. Takes objective files and stiches them all together.

What is a PE file?

## What does this executable contain and what can we see?

* .text: This section consists of machine code of all functions we would have written in C sourcefile. Lowest address.
* .data: This section consists of Global and static variables.
* BSS Segment: global un-initialized variables.
* Heap: user allocates. Grows to higher addresses.
* Stack: local variables of a function. Grows to lower addresses.

You can’t execute information stored in the stack or heap.

* rodata: This section consists of all read-only(ro) data. In our example, Hello world!!\n string is the only read-only item in the file.
* .comment:
* .eh\_frame:

How does the computer know where the functions start?

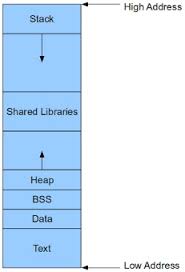
* The Symbol table. This will say hey if you want “main” or “add” look at this address.

How is each program’s memory 0x000000 -> 0xffffffff

* This is the amount of space for a 32bit architecture. Generally the very highest addresses are reserved for the kernel. Most machines have ASLR so they will randomize the first 4 places in the address.

## Alright so now that it is running what happens?

Here is memory layout of an executable



Where does the following Windows Security features come into play?

* DEP: Can mark stack and heap as non-executable
* ASLR: Windows Vista, 2008 server, and Windows 7 offer yet another built-int security technique (not new, but new for the Windows OS), which randomizes the base addresses of executables, dll’s, stack and heap in a process’s address space (in fact, it will load the system images into 1 out of 256 random slots, it will randomize the stack for each thread, and it will randomize the heap as well). This technique is called ASLR (Address Space Layout Randomization).

This technique was used in the famous Animated Cursor Handling Vulnerability Exploit (MS Advisory 935423) from march 2007, discovered by Alex Sotirov. The following links explain how this bug was found and exploited : http://archive.codebreakers-journal.com/content/view/284/27/ – ani-notes.pdf – http://www.phreedom.org/research/vulnerabilities/ani-header/ and Metasploit- Exploiting the ANI vulnerability on Vista

This particular exploit was believed to be the first exploit that bypasses ASLR on Vista (and, while breaking protection mechanisms, also bypasses /GS – well, in fact, because the ANI header data is read into a structure, there was no stack cookie :-)).

The idea behind this technique is quite clever. ASLR will randomize only part of the address. If you look at the base addresses of the loaded modules after rebooting your Vista box, you’ll notice that only the high order bytes of an address are randomized. When an address is saved in memory, take for example 0x12345678, it is stored like this :

LOW HIGH

87 65 43 21

When ASLR is enabled, Only “43” and “21” would be randomized. Under certain circumstances, this could allow a hacker to exploit / trigger arbitrary code execution.

Imagine you are exploiting a bug that allows you to overwrite saved EIP. The original saved EIP is placed on the stack by the operating system. If ASLR is enabled, the correct ASLR randomized address will be placed on the stack. Let’s say saved EIP is 0x12345678 (where 0x1234 is the randomized part of the address, and 5678 points to the actual saved EIP). What if we could find some interesting code (such as jump esp, or something else useful) in the addres space 0x1234XXXX (where 1234 is randomized, but hey – the OS has already put those bytes on the stack)? We only need to find interesting code within the scope of the low bytes and replaced these low bytes with the corresponding bytes pointing to the address of our interesting code.

* SEH:

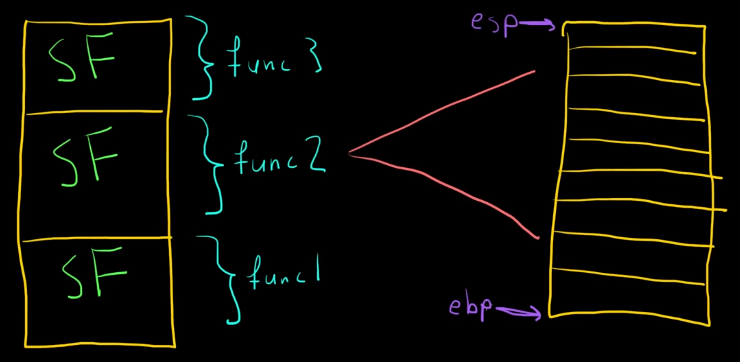
1. cause an exception. Without an exception, the SEH handler (the one you have overwritten/control) won’t kick in
2. overwrite the pointer to the next SEH record with some jumpcode (so it can jump to the shellcode)
3. overwrite the SE handler with a pointer to an instruction that will bring you back to next SEH and execute the jumpcode.
4. The shellcode should be directly after the overwritten SE Handler. Some small jumpcode contained in the overwritten “pointer to next SEH record” will jump to it).

# Assembly primer

Intel syntax is used for windows. AT&T is linux.

Memory is allocated on the heap when malloc/calloc are called or global and static variables.

Stack grows uptward. The stack grows toward lower memory addresses. Whenever a function is called all of its variables are in the stack frame.



Registers are small storage areas. They save addresses and values.

General purpose registers

* Eax
* Ecx
* Esi
* Ebx
* Edx
* Edi

Reserved

* Ebp
* Esp
* Eip
* Eflags

Segment Registers

* cs : Code Segment, Contains Starting Address of Code Segment
* ds : Data Segment, Contains Starting Address of Data Segment
* ss : Stack Segment, Contains Starting address of Stack Segment
* es : Extra Segment

At the assembly level, we will be dealing with bytes. Datatypes like char , int , long int etc., are not present at assembly level. So, these datatypes should be converted to assembly code. This is done by accessing the specific number of bytes a particular datatype in C represents.

Data:

* byte: char, 1 byte
* word: short int, 2 bytes
* dword: int, 4 bytes
* qword: long int, 8 bytes

Syntax:

* [ ] – value being pointed to is got. Not the memory address.
* Move, add, sub, mult, pop, push, etc are easy
* Jmp sets the EIP where it wants to go
* Call – push eip -> jmp func
* Leave/ret – called when function done. Destroys stack frame. Sets esp to ebp and popping ebp off stack.

# Example of reversing a simple binary

This is a half page C++ program why is there so much shit?

Libraries:

Functions:

Variables:

Execution Flow:

What it does:

Can we put this in a DEP and ASLR format and see what it looks like?