

Introduction

Preface

The purpose of this document is to provide an introduction to members of the CTF team, and others who are just interested in the topic, into Reverse Engineering (REV). This is a collaborative work in progress, written by the members of the MaaSec team that are specializing in this field. This document covers many topics that do not seem immediately related with REV but at a deeper level serve as the basis of the field (Numerical bases, computer architecture, forward engineering etc.). Because of this, aspects of this document will also prove very useful to people interested in Binary Exploitation (PWN).

If you find any missing/wrong/confusing information in this document, make sure to us know so that we can modify it. The contact information of the authors can be found in the "Contact Us" chapter.

Introduction

Reverse Engineering (REV) is the art of taking a binary file, and through disassembly, decompilation and debugging, understanding the underlying logic. REV is most commonly used in the field of Malware Analysis, but anyone can reverse anything, as long as they have access to the bytes.

The REV category of challenges in CTF competitions is usually the most difficult one. This is because obfuscation techniques have been heavily researched ever since the commercialization of computers, since their profits depend on their software being closed-source. You don't need to look for long to find an example, piracy for example is one of the main things that Music, Film and Video Game companies worry about when releasing their products. The producers don't have enough knowledge to implement good anti-piracy measures themselves, so they hire specialized companies, such as Denuvo, to do the work for them, and since their financial success depends on the effectiveness of their solutions, they have to make sure that nobody can bypass them.

With enough effort anything can be understood or get cracked, and some of the most well-known methods will be covered in the following chapters.

Basic Computing Concepts

Data and Numerical Bases/Systems

All programs are made up of data. It's up to us to give the data meaning and turn it into information. The data that makes up computer programs is in its purest form in the binary format.

Binary as you probably already know is the simplest numerical base, consisting of just 2 digits, 0 (zero) and 1 (one).

The amount of possible elements that are able to make up a valid number inside an arbitrary system, determines the base of that specific numerical system. The binary system has a base of 2 (two) since $|\{0,1\}| = 2$.

Binary data is impossible for humans to read at a large scale, so other systems are usually used. Base 10 (Decimal) ($|\{0,1,2,3,4,5,6,7,8,9\}| = 10$) has been the cornerstone of mathematics ever since the beginning. It's not really [the best base](#) but we have used it extensively, since it maps unto the amount of fingers that the median person has.

Base 16 (HEXadecimal) ($|\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}| = 16$) is a golden standard in the sphere of computing, since two HEX characters are able to represent an entire byte.

Ex. $0x00 = 0$ / $0x05 = 5$ / $0x0F = 15$ / $0x10 = 16$ / $0xAB = 171$ / $0xFF = 255$

There are an infinite amount of numerical systems, as long as you are able to represent an infinite amount of symbols. One of the best known big bases is Base-64 but Base-128, Base-256 and Base-518 are also possible.

Base-256 (AKA ASCII) is technically one of the most used bases, since all keyboards with the English alphabet depend on it, but using it for computations and mathematics is not ideal.

Translating from Base-X to Base-Y

Since numerical bases are just different ways to represent numbers, we must also be able to turn numbers from an arbitrary base into numbers of another arbitrary base. There is a simple formula that is able to do this:

$$N_B = \sum_{i=0}^n d_i \cdot A^i$$

- N_B : Number in new base
- d_i : Digit at position (i) (starting from the rightmost digit, (i = 0))
- A: The original base
- n: Highest position index (leftmost digit)

No data is lost or gained when using a certain system over another. The only thing that changes is the amount of symbols that the reader must know to interpret the data and the amount of indexes required to form the number.

This process of translating bases is best known as encoding, which is wholly different from encryption, where the original form of the data is only recoverable using a secret.

Data Sizes

Since the size of primitive data types, such as int, char and float are system dependent, we use different terms to represent the size of a type. In the binary numerical system a bit is the smallest possible unit, with which we can represent a number (0 or 1).

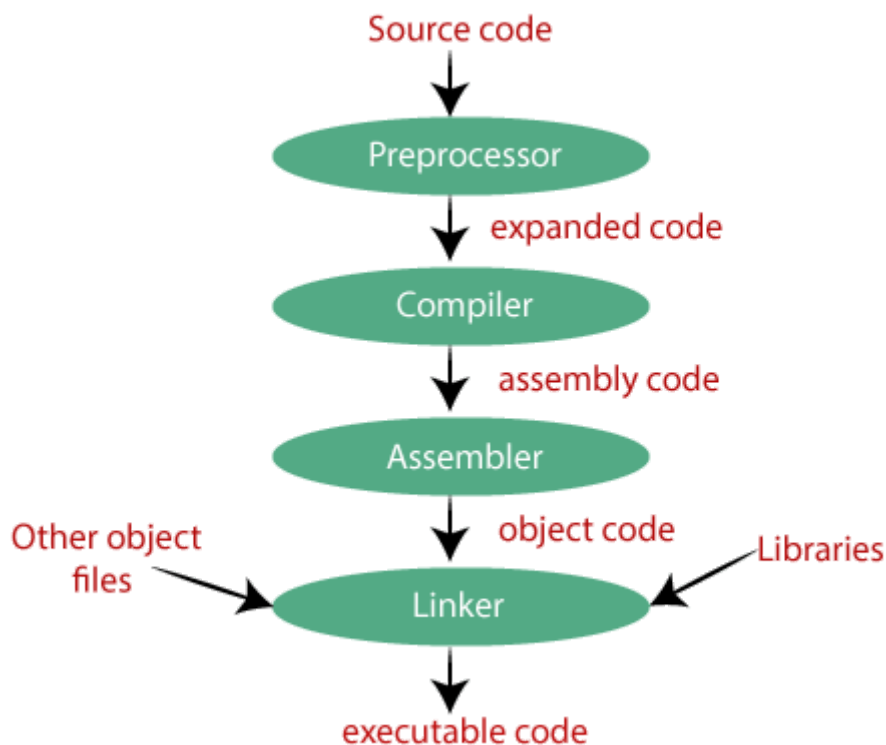
A collection of 8 bits is called a byte and has a maximum value of 255 (0-255, meaning 256 possible numbers). Byte notation is usually preferred over bit notation, since in larger data types/structures it is difficult for a human to interpret.

A half-word is a data type of 16 bits (2 bytes, max 65_535), single words (WORDS) of 32 bits (4 bytes, max 4_294_967_295) and Double Words (DWORDS) 64-bits (8 bytes, max 18_446_744_073_709_551_616).

These terms are usually used when talking about the architecture of a computer. Older computers used 32-bit registers which meant that the maximum amount of bits that a register could hold was a WORD. The address space is also dependent on the architecture, since a register holds the Instruction Pointer.

Compilation

When source code is written by a programmer (or LLM), the computer is not able to parse the source code in any meaningful way. This is because source code is no different than any other text document to the processor. The process of turning source code into executable code is called compilation but it really consists of multiple steps.



The steps of the compilation process, image from [medium](#).

Compilation (Again)

The compilation step involves turning source code into assembly code. Assembly code is a lower-level language than most others, but is not the lowest-level that we can reach.

Turning source code into assembly code requires a Compiler, since doing it manually would be impossible for big applications. Each language has its own library of compilers, and interpreters (we will cover this later), so we will take the C language as an example.

The LLVM and CLANG are examples of programs that parse the source code in such a way that it allows for easier translation to assembly. Provided a C source code file, these programs look at code patterns, such as memory allocation or mathematical operations, and are able to logically create a sequence of assembly instructions that perform the given functions. Simple compilers can be written in a couple hundred lines of code, but the specifics are outside the scope of this document.

During the process of compilation, the functionality of the program is preserved, but unique identifiers of variables and functions are lost. This makes reading assembly more challenging, since we can only use the instructions as reference. Assembly uses registers to store and operate upon data, so C code like:

```
int money = 123;
```

is translated to:

```
MOV EAX, 123
```

Here, the identifier of the "money" variable is lost, and the value of it is stored in a memory register of the compiler's choice.

Basic introduction to ISAs and Assembly

There are many different flavors of assembly, such as x86-64 which is the standard in modern computers, ARM which has many benefits over x86-64 but still has not caught on in the mainstream, and MIPS which is not talked about much. These flavors are called Instruction Set Architectures (ISAs) in more formal language and list what operations are available to a system, how to use them, and what they do. The ISA of a computer is determined by the processor. Intel is the creator of the x86-64 ISA so most of their processors use it. Many of the Apple products used to use ARM CPUs, before they switched to producing their own processors.

Another way to differentiate between ISAs is by looking at the complexity of their Instruction Sets. Reduced Instruction Set Computers (RISC) use fewer and simpler instructions, which makes them simpler and faster, but makes code more verbose, since every single step needs to be written by the developer or the compiler. Complex Instruction Set Computers (CISC) use more, and more complicated instructions. As a result, many functions can be represented in a single operation, making the code more readable, but increasing the complexity of the architecture and requiring more clock cycles to perform certain operations.

Assembly instructions consist of operations and operands, and the amount of operands depends on the operation itself. The maximum amount of operands an instruction can have, in the x86-64 ISA, is 3, but most use 1 or 2.

```
MNEMONIC DESTINATION_OPERAND SOURCE_OPERAND OPTIONAL_OPERAND
```

Assembly

The Assembly stage of compilation is the second stage, and it uses as input the assembly code generated by the compiler to produce object code. Object code is a lower level version of the assembly code in binary format, which will be used to produce the executable.

In this stage the assembly instructions are turned into their binary representations.

Linker

The Linker stage uses the object file that was generated during the Assembly stage and combines it with other object files and libraries to produce the final executable file. The final product will be a .exe file or an ELF file and can be used by the operating system to perform the functionalities determined in the source code.

Linking can either be dynamic or static. This determines the amount of external resources that are included alongside the application's source code.

Dynamic linking includes references to included files that have to be present on the system during runtime. The burden of having the files present is on the user, so it requires more effort on their end, while keeping file sizes low.

Static linking includes the files themselves into the executable. As a result file sizes are greater but allow users to immediately execute the program, without a need for additional work.

When reversing programs, static linking causes many problems to us reversers. Golang compilers for example, statically compile programs by default, meaning that analysis and debugging is much slower and much more arduous.

Computer Memory

Data and a program's code both need to be stored on the system to allow the processor to perform operations. Computers use a multi-layer system of caches and cold memory to improve performance, with the fastest memory type being CPU registers.

Registers

Registers exist inside the processor and are used for high-performance storage and operations. The size of the register depends, as previously mentioned, on the architecture of the CPU, with most modern registers holding 64 bits of data.

Different registers are used for different reasons and knowing the purpose of each one, is required, when analyzing a binary file.

General Purpose Registers

General Purpose Registers (GPRs) hold data that the function uses to complete its operations. When initializing a variable using a given value or operating upon values, general purpose registers hold those values.

They can hold both data and addresses, which means that they can serve as pointers to other locations in memory too. Since the processor does not differentiate between pointers and data, it is up to the developer and the reverser to decide which type it is.

We can break GPRs (not exclusively, but mainly) into sub-registers for more fine-grained control over the data. The 64-bit RAX register for example, consists of the EAX register, which holds the lower 32-bits of the RAX register, the AX register, which holds the lower 16-bits and AL the 8-bits, whereas AH, the following 8-bits, meaning bits 8-15.

Monikers					Description
64-bit	32-bit	16-bit	8 high bits of lower 16 bits	8-bit	
RAX	EAX	AX	AH	AL	Accumulator
RBX	EBX	BX	BH	BL	Base
RCX	ECX	CX	CH	CL	Counter
RDX	EDX	DX	DH	DL	Data (commonly extends the A register)
RSI	ESI	SI	N/A	SIL	Source index for string operations
RDI	EDI	DI	N/A	DIL	Destination index for string operations
RSP	ESP	SP	N/A	SPL	Stack Pointer
RBP	EBP	BP	N/A	BPL	Base Pointer (meant for stack frames)
R8	R8D	R8W	N/A	R8B	General purpose
R9	R9D	R9W	N/A	R9B	General purpose
R10	R10D	R10W	N/A	R10B	General purpose
R11	R11D	R11W	N/A	R11B	General purpose
R12	R12D	R12W	N/A	R12B	General purpose
R13	R13D	R13W	N/A	R13B	General purpose
R14	R14D	R14W	N/A	R14B	General purpose
R15	R15D	R15W	N/A	R15B	General purpose

List of GPRs, their sub-registers and their main purpose, image by osdev.org

Pointer Registers

The Instruction Pointer (RIP) is the main pointer register in the processor and it determines what instruction the processor will execute next. The processor looks at the pointer that it holds, and fetches, decodes and executes the instruction that it points to. These three steps make up the "Instruction Cycle" of the computer.

We can also consider the Register Stack Pointer (RSP) and Register Base Pointer (RBP) types of pointer registers. RSP holds the address of the location of the top of the stack. When we PUSH or POP data onto the stack, we use the RSP register as a reference. A programmer can manually increase or decrease the value of RSP, compared to the base, which allocates and de-allocates memory on the stack, for future use.

RBP holds the value of the base of the stack (Base / Frame pointer). We use the RBP register to allow for the creation of multiple sub-stacks (when calling functions for example), by changing the address that it points to. Additionally, we can consider the RBP register as a kind of anchor-point, with which we can point to any element on the stack at a given point, without the need to take shifts of RSP into consideration.

NOTE

RSP and RBP are NOT Pointer Registers, they are GPRs, but it aids in understanding to consider them the same as the RIP register.

Flags Register

Sometimes the processor needs to send a signal to itself in the future, to prevent unwanted behaviors, or increase performance. For example, when adding two numbers that result in a number that is greater than the maximum value that the architecture is able to represent, the processor will activate the Carry Flag, that signifies that the resulting value is not going to be the same as the expected one.

There are 22 bits worth of flags in the RFLAGS register, but the main ones are:

Name	Purpose
Carry Flag (CF)	Signal Overflow
Parity Flag (PF)	Indicate that result of calculation is even
Zero Flag (ZF)	Indicate that result of calculation is zero
Sign Flag (SF)	Indicate that result of calculation is negative

Each of these flags occupy just a single bit in the RFLAGS register, and there are many more flags that have a unique task. More information regarding flags and registers can be found [here](#).

Register Calling Conventions

When writing assembly, or compiling source code, it is useful to have a few guidelines that everyone follows. The most important family of guidelines that exists is the Calling Convention set of rules, when calling or getting called by a function.

Just like in source code, we sometimes want to pass values to a callee function. The GPRs allow for 6 values to be passed, and the rest need to be put on the stack, as will be analysed in the Stack sub-chapter. The registers follow the RDI, RSI, RDX, RCX, R8 and R9 order, meaning that RDI is the first argument that is being passed to the function and R9 the sixth.

When giving control back to the caller, at the end of a function, the return value is stored in the RAX register. So, if we

There are also conventions regarding the way that the value of a register should be saved onto the stack, but that is not required knowledge for the time being.

```
<equation>:
    endbr64
    push    %rbp
```



```

    mov    %rsp,%rbp
    mov    %edi,-0x14(%rbp)
    mov    %esi,-0x18(%rbp)
    mov    -0x14(%rbp),%eax ; Move a into the EAX register
    imul   %eax,%eax      ; Multiply a by itself to get the square
    mov    %eax,%edx
    mov    -0x18(%rbp),%eax ; Retrieve the value of b
    add    %edx,%eax      ; Add b to EAX, which now contains a^2
    mov    %eax,-0x4(%rbp)
    mov    -0x4(%rbp),%eax ; Store the resulting value in EAX, so that the
caller (main) is able to use it
    pop    %rbp
    ret

<main>:
    endbr64
    push   %rbp
    mov    %rsp,%rbp
    sub    $0x20,%rsp
    mov    %edi,-0x14(%rbp)
    mov    %rsi,-0x20(%rbp)
    movl   $0x33,-0xc(%rbp) ; Here, a gets initialized
    movl   $0xc,-0x8(%rbp)  ; Here, b gets initialized
    mov    -0x8(%rbp),%edx  ; b is stored inside the EDX register
    mov    -0xc(%rbp),%eax  ; a is stored inside the EAX register
    mov    %edx,%esi       ; b is moved into ESI, second argument register
    mov    %eax,%edi       ; a is moved into EDI, which is the first
argument
    call   1129 <equation>  ; Call the equation function
    mov    %eax,-0x4(%rbp) ; Store the result of the function on the stack
    mov    $0x0,%eax
    leave
    ret

```

Stack

The stack is a structure inside Random Access Memory (RAM) that is used by the program to store local variables for a limited amount of time. As previously mentioned, there can be multiple sub-stacks that are described by RBP (Base) and RSP (Top).

Imagining the stack as a stack data-structure is not entirely correct. PUSHing (adding) and POPing (removing) elements from the stack is one of the most used features of it. Besides that, we are also able to specify locations on it directly. An addressable location on the stack is able to store as many bits as the architecture of the processor. x86-64 is a 64-bit architecture, so each location on the stack is able to hold 64-bits worth of data. This means that it is ideal for static size, and small data structures, with dynamic size, and larger data structure moving to the heap.

The stack is used when the amount of data that we need to store is more than the registers are able to provide. This process is called stack-spilling and introduces minor slow-downs, since registers are faster than RAM, but is quite often unavoidable.

```
0x7fffffffda90: 0xffffdac0      0x00007fff      0x5555517c      0x00005555
0x7fffffffdaa0: 0xffffdbd8      0x00007fff      0x00000064      0x00000001
0x7fffffffdad0: 0x00001000      !0x00000033     !0x0000000c     0x00005555
0x7fffffffdae0: 0x00000001      0x00000000      0xf7d9cd90      0x00007fff
0x7fffffffdad0: 0x00000000      0x00000000      0x5555514c      0x00005555
```

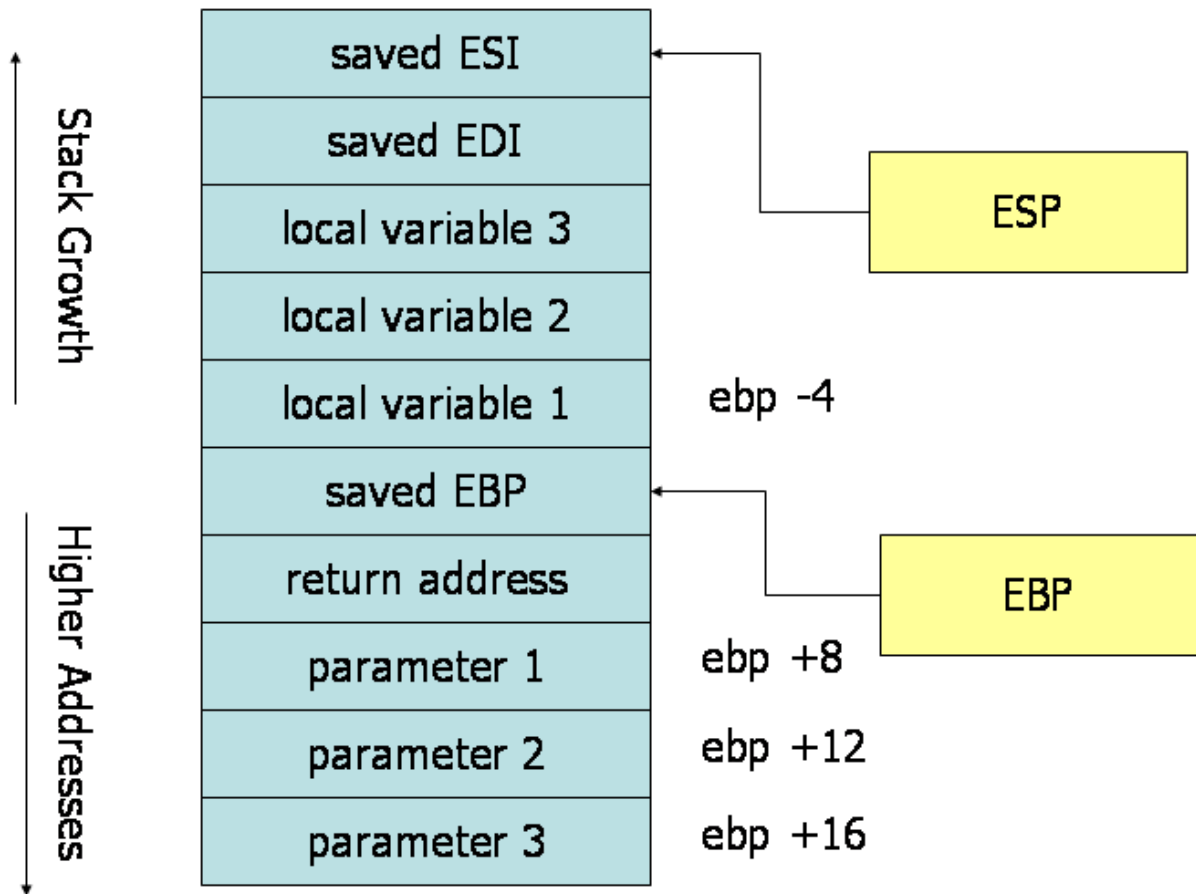
We can see that 0x33 is stored in dab4 and 0xc in dab8, which are locations on the stack.

Example of direct indexing on the stack

Consider this scenario, you push a value onto the stack and after it push 4 more values, moving the RSP away from it. When you want to access the value that you pushed onto the stack, you have to pop off each of the 4 following element to reach it. This will of course overwrite those elements, and maintaining them would be a big hassle. Instead of POPing 4 times, we can directly specify the location manually.

```
PUSH 1
PUSH 2
PUSH 3
PUSH 4
PUSH 5
```

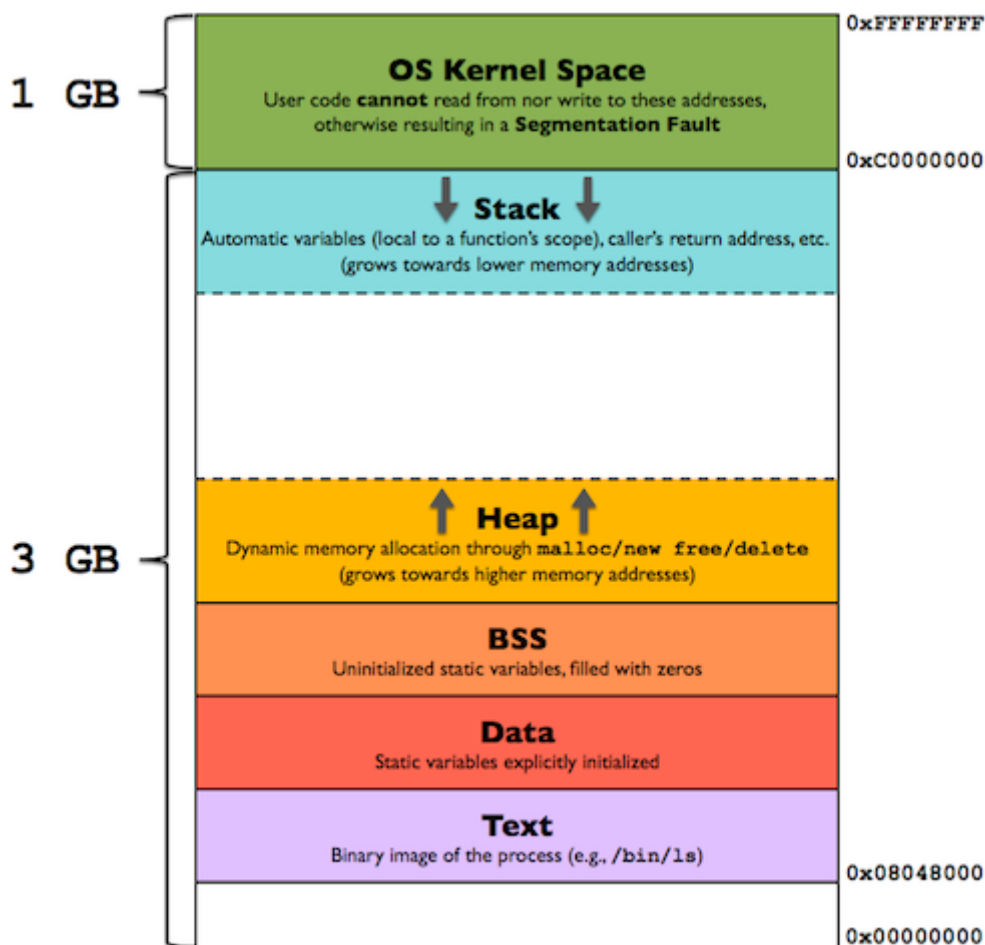
```
MOV RAX, [RSP + 32] ; Load 1 into RAX, offset is 4 x 8 = 32
```



Stack from a system with 32-bit architecture, image from virginia.edu.

Heap

Heap memory is also a memory structure inside RAM, but its implementation and usage differs slightly, when compared to the stack. Heap memory does not allow for PUSHing and POPing, but only direct indexing. When adding data to the heap, the computer needs to first check if there is enough space to hold the data. If that is not the case, memory is allocated, which is an expensive operation. Access and write times are the same between the stack and the heap, but the allocation time overhead makes it less appealing in cases where high-performance is important.



Memory Layout of Programs, image from [stackoverflow](https://stackoverflow.com).

Source Code

Being proficient in forward engineering is required for reverse engineering. Many of the standard programming patterns have an equivalent disassembled version, and this aids us in the process of understanding a binary file.

Variables

To store values in memory, we use variables. The processor can not differentiate between data types, since it sees everything as numbers (bits), so types are mainly used for programmers to be better able to understand and debug programs.

Variables can either be primitive or user defined (structures).

Primitives

Primitive types just describe how much data can fit inside the variable.

Name	Size (bytes)	Notes
short	2	
int	4	
long	8	

Name	Size (bytes)	Notes
float	4	Floating point numbers
double	8	Floating point numbers
char	1	

```

short a = 1;
int b = 0x50; // Assign a value in hexadecimal format to the variable,
does not change the functionality in any way
long c = 0b1000000; // We can also use the binary format, once again, it
does not change the functionality of it.
float d = 3.13;
double e = 222222.222222;
char f = 'f'; // Characters are just numbers, so this is the same as
assigning it the value of 102
char g = 64;

```

Symbol	Operation	Example
+	Addition	1 + 3
-	Subtraction	1 - 3
/	Division	6 / 2
*	Multiplication	7 * 7
%	Modulo	5 % 3 = 2
=	Assignment	int x = 5 + 9
==	Check Equality	int x = 5; int y = 5; if (x == 5) ...
&	Bitwise AND	0b01 & 0b11 = 0b01
	Bitwise OR	0b10 0b01 = 0b11
^	Bitwise XOR	0b11 0b10 = 0b01
~	Bitwise NOT	int x = 0b010; int y = ~x; // y = 0b101
>>	Bitwise Right Shift	0b0101 >> 1 = 0b0010
<<	Bitwise Left Shift	0b0101 << 1 = 0b1010

Structures

It is often advantageous to be able to group multiple variables together into one unit. In the C language we are able to define our own custom structures that contain an arbitrary amount of arbitrary data types.

```
struct person {  
    int age;  
    float height;  
    char* name;  
};
```

Pointers

Pointers seem to be the bane of most programmers, but in reality they are just variables that hold the location of a location in memory. This can be used for standalone variables, but is the basis of arrays, and as we will see, Strings.

```
#include <stddef.h>  
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    int x[4] = { 1,2,3,4 };  
    int *pointer = x;  
  
    size_t length = sizeof(x) / sizeof(x[0]);  
    int sum = 0;  
    for (int i = 0; i < length; i++){  
        sum += *pointer;  
        pointer += 1;  
    }  
  
    printf("%d", sum);  
  
    return 0;  
}
```

Strings

Strings are a unique type of variable, since they are closer to lists than any primitive type. A pointer object points to the first character of the string, which is usually stored on the heap, and the computer keeps moving through it until it finds a string terminating byte. String terminating bytes (NULL bytes - 0x00) consist of only 0s, and signal to the computer that there are going to be no more characters after it.

Conditionals

When we want the program to perform an operation under only certain conditions we can use a conditional to allow or disallow the execution of the nested lines.

```
int c = 15;

if (c % 5 == 0){
    printf("Number is divisible by 5");
}
```

```
int age = 21;
int money = 10;
int item_price = 7;

if ( age >= 18 && money >= item_price ){
    printf("You can buy this item");
    money -= item_price;
}
else if ( age < 18 ){
    printf("You are too young");
}
else {
    printf("Get your money up");
}
```

Symbol	Logical Operation	Example
&&	Logical AND	X && Y
	Logical OR	X Y
!	Logical NOT	!X

Loops

When we want to perform an operation multiple times, we can use a loop, instead of duplicating the code. The two main types of loops are For loops and While loops, but as we will see in further chapters, they are both While loops.

```
int sum;
for (int i = 0; i < 10; i++){
    sum += i;
}
```

```
int i, sum;
while (i < 10){
    sum += 1;
    i += 1; // or i++;
}
```

Functions

Functions allow us to define a set of instructions that will be performed, whenever it is called. We can pass parameters to functions, through function arguments, but it is not required. We can also retrieve the result of the function, but yet again, that is not required.

```
/*  
In the signature of the function:  
    long determines the return type, could be any defined data type,  
including user-defined  
    calculateSumSquared is the name of the function  
    int a is the data type of the first argument and its name  
    int b is the data type of the second argument and its name  
*/  
long calculateSumSquared(int a, int b){  
    long result = (a + b)*(a + b);  
    return result;  
}  
  
int main(){  
    long sumS = calculateSumSquare(1,2); // sum = 9  
}
```

```
void sayHello(){  
    printf("Hello");  
    // No return required, since void  
}
```

Reversing

Tools

Due to the niche nature of Reverse Engineering, not many tools exist out there that are aimed at simplifying the work-process. The tools are also incredibly complex, so only big organizations (such as the NSA...) are able to produce them. To reverse engineer a program we just need a way to statically and dynamically analyse them.

Static Analysis

Static analysis is the process of taking an executable file and analyzing its contents without executing the file itself. The result is the functional side of the program (assembly instructions) alongside any possible hard coded data values that were stored alongside them (mainly strings).

If you are using Linux on your system, you will likely have access to the "objdump" tool. Using:

```
objdump -d [PATH_TO_EXECUTABLE]
```

will print the disassembled view of the chosen binary. Don't worry if you don't understand it just yet, we will get to that point.

Objdump is nice to work with for small projects (handful of simple functions), but gives us no interactability with the file itself. Programs like [Ghidra](#) address this issue, by providing a whole suite of tools for reverse engineers.

The program itself is intended for Malware Analysis, but a binary is a binary.

One of the key features that Ghidra allows us to use, is the Decompiler. Instead of looking at assembly instructions manually, the decompilation software parses the disassembly and creates high-level pseudocode with equivalent functionality. We are then able to read that pseudocode to get a better understanding of the underlying logic of the program.

NOTE - Decompilation

Decompilation is not a perfect process, and a lot of information is lost during it. It is nice to work with, but investing into understanding assembly, will prove beneficial in the long run.

NOTE - Choice of Disassembler/Decompiler

There are a few other Disassemblers/Decompilers, namely [IDA](#), [Binja](#) and [Radare](#). Which one you choose is up to you, since they are all quite equal in terms of quality. Ghidra has one of the nicest decompilers, but IDA has one of the nicest disassemblers. Binary Ninja (Binja) is both a dynamic (we talk about this in the next chapter) and static analysis tool with both being of relatively high quality. People tend to fight over tribalism, instead of the actual quality of the products themselves, so use them all and decide for yourself.

Personally I use the Ghidra and Binja stack on both Linux and Windows, and have had no troubles. There are a few complementary tools that I do use on Linux, that I will talk about during the Dynamic Analysis chapter.

Dynamic Analysis

Instead of just relying on the phenotype of the code during analysis, we can see what the state of the CPU and memory is like during execution. By using a debugger we can stop the execution of the program at user-defined spots and step through the assembly instruction by instruction to get a better grasp of how data is operated upon.

All of the tools referenced in the Static Analysis chapter also have the capacity to debug or attach debuggers, but using standalone debuggers is usually the better option. On windows [x64dbg](#) is the standard, but Binja and [ollydbg\(outdated\)](#) can also be used. Since x64dbg is Windows specific, we have to use another debugger on Linux. [GDB](#) can do everything x64dbg can, but with a simpler UI. Using the [PWNDGB](#) extension for GDB further enhances the capabilities of the debugger, making it the ideal tool for Reversing and PWNing applications.

TANGENT - When to statically, and when to dynamically analyse?

There is no real guide-line on how to decide. In my personal experience statically analyzing at first and then dynamically has granted me fine results. There are people who swear by only using static analysis, and others the same but for dynamic. It is up to the individual reverser to decide when and if they use either option.

For beginners I would suggest following my approach and then tailoring it to create their own workflow.

Memory Scanning

A memory scanner is a feature that usually exists in the aforementioned tools in some barebones way. Dedicated memory scanners give us a lot more freedom over the process of looking at and searching for values in the memory. The best known memory scanner is [CheatEngine](#) which is heavily used by the video-game modding community. Memory scanners can also be used for non-video game programs but are not necessary.

When we know that a certain numerical value exists inside the program, but do not know the exact location of it, we can skim through the entire memory and find the address where it is stored. CheatEngine also has a disassembler built-in and that means that we can find both the exact locations in memory, where the value is stored, but also see how it is used by the program.

Certain challenges require CheatEngine to be solved, but those cases are few.

Scripting

Scripting is the forgotten about child of the reverse engineering world. There is no good way to teach scripting, since each use-case is unique. Using scripts to emulate the behavior of the program, or to filter data are two common ones, but there is an infinite amount of possibilities. Platforms like Ghidra allow for embedded scripting, using their custom API, but any scripting language can be used. Python is the most used language for scripting, but Golang and Lua are also good choices. The python [PWNTOOLS](#) library implements a lot of features that are impossible to live without once you get used to them. It is useful for both REV and PWN challenges, and is something that everyone should get, even surface level, experience in.

x86-64 Assembly Basics

Just as source code is comprised of loops, conditionals, functions and basic mathematical operation, so assembly is comprised of conditionals, functions, data movement and basic mathematical operations.

NOTE - Why are there no loops in assembly? - IMPORTANT

As we mentioned during the explanation of loops, a for loop is just a while loop with nicer syntax. A while loop on the other hand is just a conditional that is evaluated each time we reach a certain line in the code. Assembly languages have a unique feature where we are able to jump to a specific code address, and that means that we can create a makeshift loop by jumping backwards into the code.

```
for_loop:
    cmp rax, 10                ; Compare count with 10

    jg end_loop               ; If count > 10, exit loop

    inc rax                   ; Increment count
    jmp for_loop              ; Repeat the loop

end_loop:
    ...
```

Comparisons and Jumps

In Assembly languages we need to explicitly write the instructions that compare two values. Instead of writing

```
int x = 5;
int y = 1;
if (x == y)...
```

we need to write

```
MOV 0x5, EAX
MOV 0x1, EBX
CMP EAX, EBX
JNE [address]
```

Comparisons can also be used before conditional movement instructions, as will be discussed shortly. When using the CMP operation the result of the subtraction between the first and second values is remembered, by triggering the correct flags. There are many different types of results, which allow for a lot of control over the flow of the program.

Instruction		Description	Condition Code
jmp	<i>Label</i>	Jump to label	
jmp	<i>*Operand</i>	Jump to specified location	
jz / jz	<i>Label</i>	Jump if equal/zero	ZF
jne / jnz	<i>Label</i>	Jump if not equal/nonzero	~ZF
js	<i>Label</i>	Jump if negative	SF
jns	<i>Label</i>	Jump if nonnegative	~SF
jg / jnle	<i>Label</i>	Jump if greater (signed)	~(SF^OF)&~ZF
jge / jnl	<i>Label</i>	Jump if greater or equal (signed)	~(SF^OF)
jl / jnge	<i>Label</i>	Jump if less (signed)	SF^OF
jle / jng	<i>Label</i>	Jump if less or equal	(SF^OF) ZF
ja / jnbe	<i>Label</i>	Jump if above (unsigned)	~CF&~ZF
jae / jnb	<i>Label</i>	Jump if above or equal (unsigned)	~CF
jb / jnae	<i>Label</i>	Jump if below (unsigned)	CF
jbe / jna	<i>Label</i>	Jump if below or equal (unsigned)	CF ZF

Types of Jumps that can be performed, depending on the result of the CMP call, Image from [Brown University](#).

We can also use the TEST operation, which uses a Bitwise AND, instead of a subtraction.

Data Movement

To move values into registers, between registers or from/to a register to/from RAM we need to use a MOV operation.

```
MOV SOURCE, DESTINATION
```

There are multiple types of movement instructions and they all do slightly different things. Instead of performing manual comparisons and then using jumps for branching, we can use the 'c' prefix with the appropriate suffix to achieve the same behavior in one line, if the following instruction would be a move.

Instruction		Description
cmove / cmovz	<i>S, D</i>	Move if equal/zero
cmovne / cmovnz	<i>S, D</i>	Move if not equal/nonzero
cmovs	<i>S, D</i>	Move if negative
cmovns	<i>S, D</i>	Move if nonnegative
cmovg / cmovnl	<i>S, D</i>	Move if greater (signed)
cmovge / cmovnl	<i>S, D</i>	Move if greater or equal (signed)
cmovl / cmovnge	<i>S, D</i>	Move if less (signed)
cmovle / cmovng	<i>S, D</i>	Move if less or equal
cmova / cmovnbe	<i>S, D</i>	Move if above (unsigned)
cmovae / cmovnb	<i>S, D</i>	Move if above or equal (unsigned)
cmovb / cmovnae	<i>S, D</i>	Move if below (unsigned)
cmovbe / cmovna	<i>S, D</i>	Move if below or equal (unsigned)

Types of conditional movement operations, Image from [Brown University](#).

Mathematical Operations

Most mathematical operations that are present in the c language, can also be performed in assembly languages, quite succinctly. For example, when adding two values, instead of using the '+' symbol, we use the "ADD" operation, with the two values are operands.

3.2.2 Binary Operations

Instruction		Description
leaq	<i>S, D</i>	Load effective address of source into destination
add	<i>S, D</i>	Add source to destination
sub	<i>S, D</i>	Subtract source from destination
imul	<i>S, D</i>	Multiply destination by source
xor	<i>S, D</i>	Bitwise XOR destination by source
or	<i>S, D</i>	Bitwise OR destination by source
and	<i>S, D</i>	Bitwise AND destination by source

3.2.3 Shift Operations

Instruction		Description
sarl / shl	<i>k, D</i>	Left shift destination by <i>k</i> bits
sar	<i>k, D</i>	Arithmetic right shift destination by <i>k</i> bits
shr	<i>k, D</i>	Logical right shift destination by <i>k</i> bits

Some common mathematical operations in x86-64, Image from [Brown University](#).

Functions


The basic structure of functions, is quite simple.

The processor changes the value stored inside the RIP register to the first line of the called function. The return address of the caller function is PUSHed on to the stack and the function is executed. At the end a RET call is made to return the control to the caller function, after POPing the return address from the stack.

We are also able to create child processes and threads that execute functions in parallel with the main and other child processes/threads. This will be covered in future chapters, since parallel processing is quite common in larger applications.

Appendix A - Community

Contact Us

	Name	Social-Media HyperLinks
	Dimitrios Tsiplakis	GitHub LinkedIn johnnnathan@proton.me

Additional Tangents

Using AI to Reverse Programs

In your reverse engineering journey you will definitely come across a function that you cannot understand, no matter the hours that you spend on it. I myself, and many others, have asked LLMs for help in such cases. LLMs are, in my opinion, detrimental to your development as a reverser, not because LLMs complete the job instead of you, but because they are not assembly models, they are language models. The quality of the responses of the models are based on the quality of the content that the developers used while training it. Pure assembly is amongst the least used programming languages, and so, the possible amount of training data is limited, compared to a language like python. It is highly likely that a model will give you confusing or false results when querying it, so do your best to not depend on them.

Notes

When talking about specifics of assembly, and the ISA is not specified assume that it is x86-64. In the first few chapters we specify it, but this is not the case in further chapters.

Most source code snippets will be written in C. It's what most apps use and the language I like the most, fight me.

Any provided binaries will be ELF's. In the future this might change. Certain referenced programs might also be Linux exclusive.

Appendix B - Exercises