



Université Mohammed Premier
École Nationale des Sciences Appliquées
Oujda



End-of-Year Dissertation

Field : Sécurité Informatique et Cybersécurité

Defended on : 03/07/2024

Prepared by : ESSATAB Achraf

Binary Exploitation Technics, Evasion and Mitigation

Supervised by:

M. M. A. KOULALI

Contents

Contents.....	1
Abstract.....	2
List of Tables and Figures:.....	3
General Introduction	5
Chapter 1 Reducing the level of abstraction	6
1.1 Introduction	6
1.2 C as a Procedural Language	7
1.3 Compilers and linkers' roles in C software's development process	8
1.3.1 What is the compiler's job?	9
1.3.2 The linker's job	10
1.3.3 Summary	11
1.4 Function Calls: From High-level Statement to Machine Instructions	12
1.4.1 Overview of x86-64 Machine Architecture	14
1.4.2 Overview of the Calling Conventions of System V ABI	17
1.4.3 The Stack Evolution During Function Calls.....	19
1.4.4 Concrete Example	23
1.5 Conclusion.....	32
Chapter 2 Functions Lacking Bounds Checking	33
2.1 Introduction	33
2.2 GNU C Library.....	33
2.3 Unsafe Functions: an Example of strcpy()	34
2.3.1 The Bug in Detail	35
2.3.2 Exploit: Arbitrary Code Execution	42
2.3.3 Mitigation: access restriction to the stack.....	43
2.3.4 Evasion: Code Reuse Attack.....	45
2.3.5 Mitigation: Stack Canaries	46
2.4 Alternative Functions: an Example of strncpy().....	53
2.4.1 A proposed solution: Substituting the Flawed Function.....	54
2.5 Guidelines for secure programming	68
2.6 Conclusion.....	72
General conclusion and perspectives	73
References	74

Abstract

The security of computer systems evolves over time through an ongoing conflict between designers and attackers. Whenever an attack is discovered, there is a great opportunity to enhance the security of computer systems by addressing the cause of the attack, thus fortifying future systems against similar vulnerabilities and implementing defenses in current systems.

This work begins by eliminating unnecessary abstractions that hinder the understanding of software bugs threatening the safety of a memory region known as the stack. After gaining insights on how high-level C language statements, and especially the call construct, affect our memory region, this work will delve into low-level details (such as assembly language) to demonstrate how the absence of bounds checking in some glibc's functions can alter the control flow of a program, causing it to behave unexpectedly, when this flaw is exploited by malicious users.

Using the function strcpy() from the C standard library as an example, it will be shown how careless usage of this function by programmers can lead to exploitable bugs. The exploitation process used by attackers will be discussed. Subsequently, the security measures designers implement to mitigate these exploits will be examined, along with how attackers bypass these mitigations and the subsequent countermeasures.

This work offers a solution to strengthen programs afflicted with this bug when the source code of a vulnerable C program is unavailable. Additionally, guidelines provided by the CERT Coordination Center at Carnegie Mellon University will be presented to help prevent such vulnerabilities in the first place at the software development phase.

List of Tables and Figures:

Figure 1: Layers of abstraction.....	5
Figure 2: Function Prototype	6
Figure 3: The compiler's phases.....	8
Figure 4: C program life cycle	11
Figure 5: A function call	12
Table 6: User-controlled x86-64 registers	13
Table 7: Commonly used x86-64 instructions	15
Figure 8.a: Stack allocation in memory.....	18
Figure 8: Multiple stack frames, each per function	19
Figure 9: the stack after the call.....	20
Figure 10: Function f calls function g.....	21
Figure 11: The main function in assembly representation.....	23
Figure 12: Function f in assembly representation.....	23
Figure 13: Function g in assembly representation.....	23
Figure 14: Passing one argument.....	24
Figure 15: The call instruction.....	25
Figure 16: RBP register saved by the callee.....	26
Figure 17: Indicating a new stack frame.....	27
Figure 18: Passing integer return value.....	28
Figure 19: restoring the value of RBP.....	29
Figure 20: The ret instruction.....	30
Figure 21: The end of the call sequence.....	30
Table 22: Popular vulnerable functions.....	33
Figure 22: A program using strcpy.....	36
Figure 23: Corrupted caller's stack frame.....	37
Figure 24: Linux man page of strcpy.....	38
Figure 25: A program invoking strcpy.....	39
Figure 26: Normal execution.....	39

Figure 27: An execution with unintended input.....	39
Figure 28: Debugger view of the program.....	40
Figure 29: The corrupted return address.....	40
Figure 30: Segmentation Fault exception.....	40
Figure 31: Inner workings of shellcode exploit.....	41
Figure 32: Stack allocation using mmap.....	43
Figure 33: The memory layout of a running process.....	44
Figure 34: Code reuse attack.....	45
Figure 35: Stack frame before the corruption.....	47
Figure 36: Detected memory corruption.....	48
Figure 37: The vulnerable program.....	63
Figure 38: Trampoline implementation.....	63
Figure 39: The main program before applying the patch process.....	64
Figure 40: The workings of the patched program.....	65
Figure 41: The assembly representation of printName before the patching process.....	66
Figure 42: A substituted call instruction.....	66
Figure 43: Executing the flawed program.....	67
Figure 44: Executing the patched program.....	67
Table 45: Deprecated functions and their alternative.....	70

General Introduction

Abstractions are valuable tools in computer science; they enhance creativity and accelerate the evolution of computer systems. However, they obscure the underlying workings of computer systems, making it nearly impossible to understand how software bugs are exploited by attackers, and hence how to develop robust security countermeasures. Therefore, this work begins by attempting to strip away fundamental abstractions related to the stack memory region using simple language. After delving into the details underlying those abstractions, it will be easier to understand how functions lacking bounds checking can lead to exploitation by malicious users. We will examine the exploitation techniques used by attackers to exploit the `strcpy()` function, to give a concrete example. The primary aim of an exploitation technique is to alter the behavior of a program from its intended functionality. We will also present examples of the countermeasures devised by designers to thwart these exploits, the strategies attackers employ to bypass these mitigations, and how these bypassing methods are countered by the development of new security measures.

Software constitutes an essential component of any computer system, as it dictates to the hardware the tasks to be performed. It can be executed directly on the hardware, with restrictions [1], or interpreted by a specific runtime environment. It is almost impossible to create software that contains no bugs; The number of bugs is at least proportional to the size of the program [2], and not dependent on the programming language choice. This project focuses on addressing bugs that compromise the stack memory security in the context of applications developed in C language.

Chapter 1 Reducing the level of abstraction

1.1 Introduction

As C is a procedural language, C applications are built of a set of functions. Before delving into the issues surrounding the memory safety of a C program's stack, it is crucial to comprehend the compilation process when functions call each other and the role of the stack in this operation.

As computing hardware becomes faster and more powerful, software applications become more complex and sophisticated. New generations of computer systems spawn new generations of software that can do more powerful things than previous generations. As the software gets more sophisticated, the job of developing an application becomes more difficult. To keep the programmer from being quickly overwhelmed, it is critical that the process of programming be kept as simple as possible. Automating any part of this process (i.e., having the computer do part of the work) is a welcomed enhancement in the history of computer science.

C is a compiled high-level language, the compilation is done by tools (prebuilt software), which makes the developer more focused on his job, innovative, and comfortable than programming with an assembler language.

The use of tools such as compilers and linkers also aids developers in identifying and eliminating subtle programming errors, serving as an additional line of defense against exploitable programming bugs.

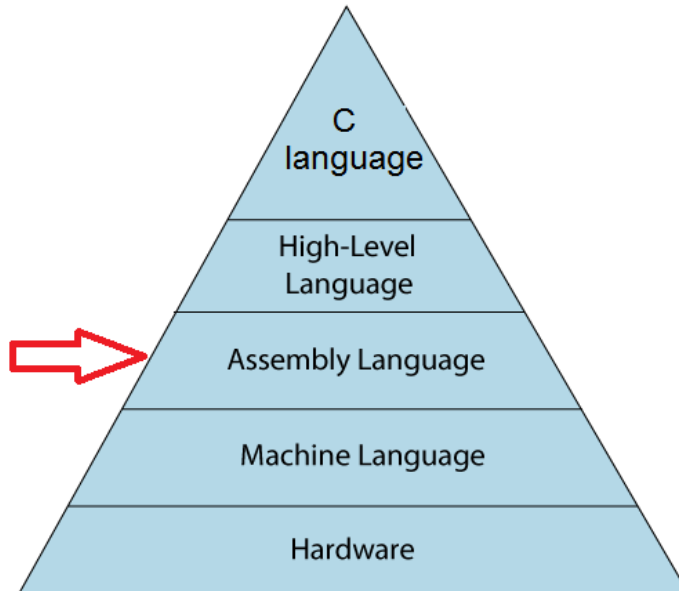


Figure 1: Layers of abstraction

C programming is considered the base for other programming languages, which is why it is known as the mother language [3]. It is considered the mother language of all modern programming languages because most of the compilers, JVMs, Kernels, etc. are written in C language, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

The goal of this chapter is to comprehend the low-level intricacies inherent in C language statements that render programming bugs related to stack memory exploitable by attackers, as well as to understand how mitigation techniques function. To achieve this goal, we will delve into the realm of assembly language abstraction.

1.2 C as a Procedural Language

C [4] is an imperative, procedural language in the ALGOL tradition. It has a static type system. In C, all executable code is contained within subroutines (also called "functions"). Function parameters are passed by value, although arrays are passed as pointers, i.e., the address of the first item in the array. Pass-by-reference is simulated in C by explicitly passing pointers to the thing being referenced.

C is an imperative procedural language, supporting structured programming, lexical variable scope, and recursion, with a static type system. It was designed to be compiled to provide low-level memory access and language constructs that map efficiently to machine instructions, all with minimal runtime support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant C program written with portability in mind can be compiled for a wide variety of computer platforms and operating systems with few changes to its source code.

A function is a named block of code that can be called from anywhere in the program, basically by another function. This design is conceived to avoid repeating a frequently used block of code in the binary (i.e., the executable) of the application. A function in C can be called either with arguments or without arguments. These functions may or may not return values to the calling functions. Hence the function prototype of a function in C is as below:

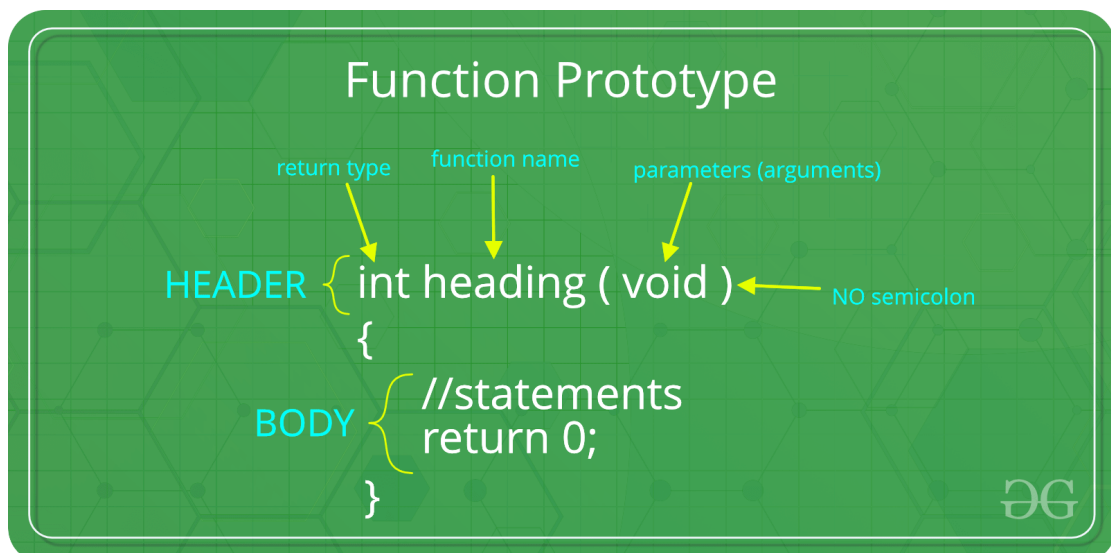


Figure 2: Function Prototype

A parameter, if exists, is either a variable passed by value, by address (e.g., pointer to the variable), or a constant value.

A function in C can return either a constant value, the value of a variable accessible by that function (i.e., in the function's scope), or a memory address.

1.3 Compilers and linkers' roles in C software's development process

High-level languages can be classified into two categories: Compiled languages and interpreted languages.

C is a compiled language; A compiled language is converted into machine code so that the processor can execute it. An interpreted language on the other hand is a language executed by an interpreter (a software), the interpreter executes instructions directly without earlier compiling a program into machine language.

Compiled programs run faster than interpreted programs because there is no interpretation overhead.

To inspect at a low level how procedure calls get translated to machine language and to set the stage for the stack memory: the core concept of this work. This part discusses, in a high level of detail, the process of converting compiled programs from a high-level language to a machine-specific language.

C programs are written in human-readable constructs forming the source code of the application. The source code is not directly executable by a computer. It takes a three-step process [5] to transform the source code into executable code. These three steps are: Preprocessing, compiling and linking.

- Preprocessing: At this stage, preprocessor directives (commands that begin with a # character) are parsed by a preprocessor which leads to modifications in the source code before being passed to the compiler.
- Compiling: This phase causes the modified source code to be compiled into binary object code. This object code is not yet executable, it lacks some critical routines and metadata.
- Linking: The object code is combined with the required supporting code to make an executable program. This step typically involves adding in any libraries that are required.

In most modern compilers, these three activities are handled by a single application, although it is possible to tell the compiler not to do certain functions. (For example, to compile but not link a program.) There are a variety of C compilers available for many different platforms. Some compilers must be purchased and some are free to use. Three of the most common are GNU GCC, Clang/LLVM and Microsoft Visual C.

GNU GCC is found on many platforms such as Linux, many flavors of UNIX, and even Windows. Clang/LLVM is available for all modern Mac OSX systems and many BSD variants. Microsoft Visual C is a core component of Microsoft's Visual Studio platform. We will use GNU gcc [6] in our use cases, mainly due to its availability on many different platforms.

1.3.1 What is the compiler's job?

In computing, a compiler [7] is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a low-level programming language (e.g., assembly language, object code, or machine code) to create an executable program.

A compiler is likely to perform some or all of the following operations, often called phases: preprocessing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization, and machine-specific code generation. Compilers generally implement these phases as modular components, promoting efficient design and correctness of transformations of source input to target output. Program faults caused by incorrect compiler behavior can be very difficult to track down and can be the source of a security flaw in the program; therefore, compiler implementers invest significant effort to ensure compiler correctness.

The compiler also comes with a set of plugins each responsible for some tasks, an example of such plugins, discussed in section 2.2.5, is one that implements a security mechanism against stack's buffer overflow attacks.

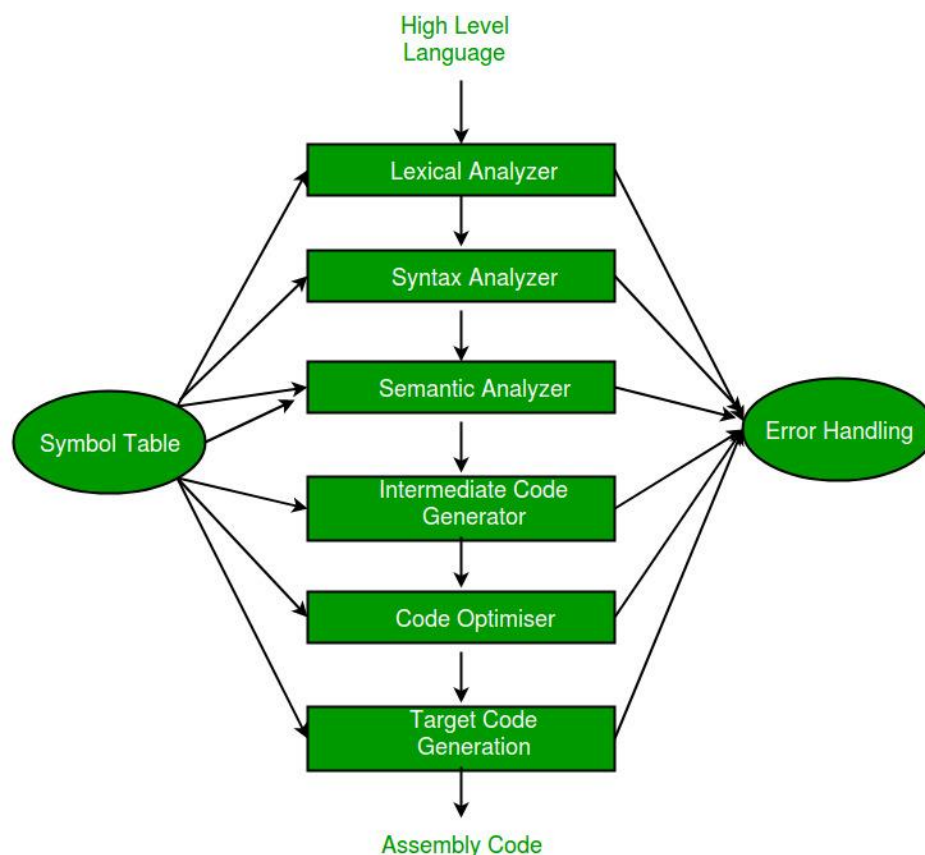


Figure 3: The compiler's phases

After the compiler has successfully finished its job, another tool called the linker kicks in. The linker's job is to combine all of the object files and produce the desired output file (e.g., typically an executable file). This process on which the linker acts, is called linking.

1.3.2 The linker's job

Linking[8] is the process of collecting and combining various pieces of code and data, from different sources (e.g., libraries and object files), into a single file that can be loaded (copied) into memory and executed. Linking can be performed at compile time, when the source code is translated into machine code, at load time, when the program is loaded into memory and executed by the loader, and even at run time, by application programs. On early computer systems, linking was performed manually. In modern systems, linking is performed automatically by programs called linkers.

Linkers play a crucial role in software development because they enable separate compilation. Instead of organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately. When we change one of these modules, we simply recompile it and relink the application, without having to recompile the other files. C programs are often linked with functions that pertain to a standard library which in most cases cover a big part of the totality of the application. The natural way to link code and data from different object files is by combining them in the same executable file, this method is called static linking.

1.3.2.1 *Static linking and dynamic linking*

Dynamic linking and static linking are two methods used to link libraries to a program, each with distinct advantages and trade-offs. In the context of a C program, understanding these methods is crucial to dig in the details behind C constructs.

Static linking involves copying all the necessary library functions into the final executable at compile time, creating a self-contained binary. This results in larger executables, as each program includes its own copy of the library code. For example, if multiple C programs use the same standard library functions, each statically linked program will have its own instance of these functions, leading to redundancy. Static linking also means that any updates to the library require recompiling all dependent programs to benefit from the changes.

Conversely, dynamic linking links the program to shared libraries at runtime. Instead of embedding the library code into the executable, the program includes references to shared library files, typically with a .so (shared object) extension on Unix-like systems. This approach significantly reduces the executable size and allows multiple programs to share a single copy of the library code, conserving memory and disk space. For a dynamically linked C program, this means that the common standard

library functions are stored in shared .so files like those provided by GNU C Library (glibc).

When a dynamically linked C program is executed, it relies on two critical sections in the final binary: the Procedure Linkage Table (PLT) and the Global Offset Table (GOT). The PLT is used for calling functions in shared libraries. Initially, when a function from a shared library is called, the PLT entry for that function redirects the call to the dynamic linker. The dynamic linker then resolves the function's address and updates the GOT with this address. Subsequent calls to the function use the resolved address directly, improving performance.

The GOT stores the addresses of global variables and functions. During program execution, the dynamic linker updates the GOT with the actual memory addresses of the shared library's functions and variables. This mechanism allows the program to access these addresses efficiently, even though their exact locations are not known until runtime.

GNU C Library (glibc) plays a pivotal role in this process. It provides essential APIs for system calls, input/output operations, memory management, and more. When a C program dynamically linked against glibc is executed, the dynamic linker loads the necessary .so files and resolves the function and variable addresses using the PLT and GOT. This ensures the program can run with the correct library code, leveraging the latest updates to the shared libraries without needing recompilation.

1.3.3 Summary

To summarize, the life cycle of a C program involves several stages from the creation of the source code to the generation of an executable file. Here are the stages that the C program passes through, in chronological order:

Writing Code: The programmer writes the C source code in a text editor or an Integrated Development Environment (IDE) and saves it with a .c extension.

Compilation: The source code is passed through a compiler (such as GCC, Clang, or MSVC) which translates it into machine-readable binary code called object code (.o files in Unix-like systems, .obj files in Windows). This process involves syntax checking, semantic analysis, and code optimization.

Preprocessing: Before compilation, the source code undergoes preprocessing. This stage involves handling preprocessor directives (such as #include and #define), which are expanded and substituted into the code.

Linking: If the program consists of multiple source files or uses external libraries, the object files need to be linked together. The linker takes care of this process by resolving references to functions and variables across different object files and libraries. It creates an executable file by combining the object code with any necessary system libraries.

Generating Executable: Finally, the linker produces an executable file (usually with a .exe extension in Windows or no extension in Unix-like systems) containing the machine code that the computer's processor can execute.

Execution: The generated executable file can be run by the operating system, and the instructions within it are executed by the CPU, performing the tasks defined in the original C program.

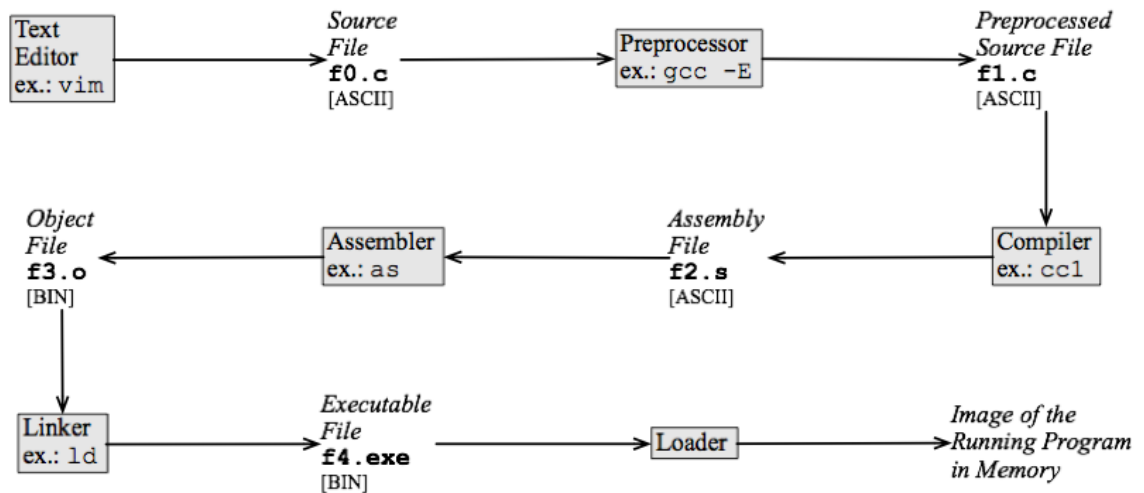


Figure 4: C program life cycle

1.4 Function Calls: From High-level Statement to Machine Instructions

Compilers and linkers operate within a framework of rules, standardized in publicly available guidelines established by trusted authorities, known as conventions. One subset of these rules is known as calling conventions, which form a significant part of a larger set of regulations called the ABI (Application Binary Interface). One of the ABI's functions is to dictate the process of translating high-level statements into machine-specific instructions.

Procedures are a key abstraction in software developed in C language. They provide a way to package code that implements some functionality with a designated set of arguments and an optional return value. This function can then be invoked from different points in a program. Understanding the effects of procedure calls on the stack requires understanding the calling convention specifications and

implementations.

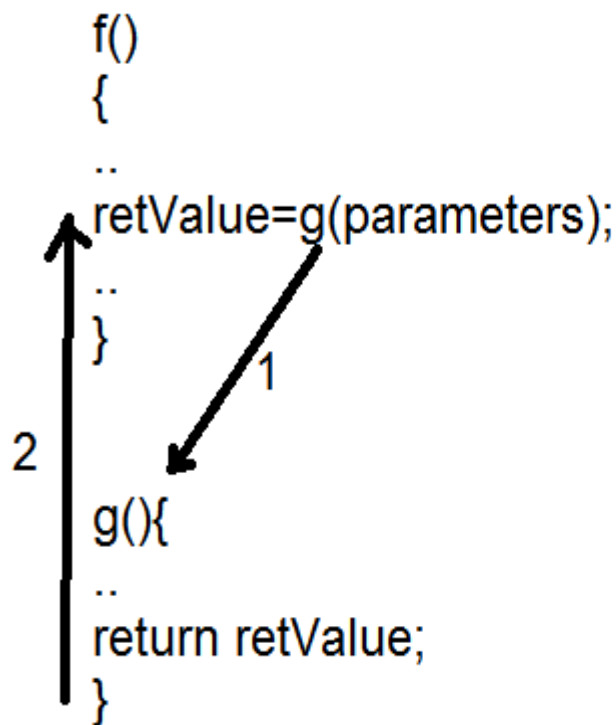


Figure 5: A function call

Many different attributes must be handled when providing machine-level support for procedures. For discussion purposes, suppose procedure `f` calls procedure `g` (see Figure 5). Control will be passed to `g`, `g` then executes and returns back to `f`. These actions involve one or more of the following mechanisms:

Passing control: The program counter (a register that keeps track of the executing instruction, see the next section) must be set to the starting address of the code for `g` upon entry and then set to the instruction in `f` following the call to `g` upon return.

Passing data: `f` must be able to provide one or more parameters to `g`, and `g` must be able to return a value back to `f`. Allocating and deallocating memory. `g` may need to allocate space for local variables when it begins and then free that storage before it returns control to `f`. The x86-64 (a computer architecture discussed in the next section) implementation of procedures involves a combination of special instructions and a set of conventions on how to use the machine resources, such as the registers and the program memory. Great effort has been made to minimize the overhead involved in invoking a procedure.

As a consequence, it follows what can be seen as a minimalist strategy, implementing only as much of the above set of mechanisms as is required for each

particular procedure. In our work, we inspected the different mechanisms. Step by step, first describing control, then data passing, and, finally, memory management.

Before having a look at the rules defining procedure calls translation to machine language instructions and reading them later in assembly language (a language that uses human-readable mnemonics) it is crucial that we have a background on the components of hardware which will execute our software.

1.4.1 Overview of x86-64 Machine Architecture

1.4.1.1 Registers

A register is the closest unit of storage to a CPU (Central Processing Unit). Storing or retrieving data from a register is faster than dealing with caches or primary memory. In the x86-64 computer architecture, there is a set of 16 64-bit general purpose registers, directly handled by user-software developers.

The following table summarizes the size and the use of the most commonly used general-purpose x86-64 registers:

64-bit	32-bit	16-bit	8-bit	Special Purpose for functions	When calling a function	When writing a function
Rax	eax	ax	ah,al	Return Value	Might be changed	Use freely
Rbx	ebx	bx	bh,bl		Will not be changed	Save before using!
Rcx	ecx	cx	ch,cl	4 th integer argument	Might be changed	Use freely
Rdx	edx	dx	dh,dl	3 rd integer argument	Might be changed	Use freely
rsi	esi	si	sil	2 nd integer argument	Might be changed	Use freely
rdi	edi	di	sil	1 st integer argument	Might be changed	Use freely
rbp	ebp	bp	bpl	Frame Pointer	Maybe Be Careful	Maybe Be Careful
rsp	esp	sp	spl	Stack Pointer	Be Very Careful!	Be Very Careful!
r8	r8d	r8w	r8b	5 th integer argument	Might be changed	Use freely
r9	r9d	r9w	r9b	6 th integer argument	Might be changed	Use freely

r10	r10d	r10w	r10b		Might be changed	Use freely
r11	r11d	r11w	r11b		Might be changed	Use freely
r12	r12d	r12w	r12b		Will not be changed	Save before using!
r13	r13d	r13w	r13b		Will not be changed	Save before using!
r14	r14d	r14w	r14b		Will not be changed	Save before using!
r15	r15d	r15w	r15b		Will not be changed	Save before using!

Table 6: User-controlled x86-64 registers

1.4.1.2 *Machine-specific instructions in assembly representation*

A CPU is capable of executing a repertory of functions called instructions, among other operating system-specific instructions, the most used instructions are either arithmetic and logic instructions, data movement instructions, or control flow instructions.

An instruction in assembly representation is an opcode (for operation code) with zero or more operands (the arguments on which the CPU operates). Each machine-specific instruction has only one equivalent in assembly language, represented in an easy-to-recognize mnemonic.

Each mnemonic opcode presented in Figure 7 represents a family of instructions[9]. Within each family, there are variants which take different argument types (registers, immediate values, or memory addresses) and/or argument sizes (byte, word, double-word, or quad-word). The former can be distinguished from the prefixes of the arguments, and the latter by an optional one-letter suffix on the mnemonic.

For example, a mov instruction that sets the value of the 64-bit %rax register to the immediate value 3 can be written as

```
movq    $3, %rax
```

Immediate operands are always prefixed by \$.

For instructions that modify one of their operands, the operand that is modified appears second. This differs from the convention used by Microsoft's and Borland's assemblers, which are commonly used on DOS and Windows.

Opcode	Description
Copying values	
mov src, dest	Copies a value from a register, immediate value or memory address to a register or memory address.
cmove %src, %dest	Copies from register %src to register %dest if the last comparison operation had the corresponding result (cmove: equality, cmovne: inequality, cmovg: greater, cmovl: less, cmovge: greater or equal, cmovle: less or equal).
cmovne %src, %dest	
cmovg %src, %dest	
cmovl %src, %dest	
cmovge %src, %dest	
cmovle %src, %dest	
Stack management	
enter \$x, \$0	Sets up a procedure’s stack frame by first pushing the current value of %rbp on to the stack, storing the current value of %rsp in %rbp, and finally decreasing %rsp to make room for x byte-sized local variables.
Leave	Removes local variables from the stack frame by restoring the old values of %rsp and %rbp.
push src	Decreases %rsp and places src at the new memory location pointed to by %rsp. Here, src can be a register, immediate value or memory address.
pop dest	Copies the value stored at the location pointed to by %rsp to dest and increases %rsp. Here, dest can be a register or memory location.
Control flow	
call target	Jump unconditionally to target and push return address (current PC + 1) onto stack.
Ret	Pop the return address off the stack and jump unconditionally to this address.
jmp target	Jump unconditionally to target, which is specified as a memory location (for example, a label).

je target	Jump to <code>target</code> if the last comparison had the corresponding result (<code>je</code> : equality; <code>jne</code> : inequality).
jne target	
<i>Arithmetic and logic</i>	
add src, dest	Add <code>src</code> to <code>dest</code> .
sub src, dest	Subtract <code>src</code> from <code>dest</code> .
imul src, dest	Multiply <code>dest</code> by <code>src</code> .
idiv divisor	Divide <code>rdx:rax</code> by <code>divisor</code> . Store quotient in <code>rax</code> and store remainder in <code>rdx</code> .
shr reg	Shift <code>reg</code> to the left or right by value in <code>cl</code> (low 8 bits of <code>rcx</code>).
shl reg	
ror src, dest	Rotate <code>dest</code> to the left or right by <code>src</code> bits.
cmp src, dest	Set flags corresponding to whether <code>dest</code> is less than, equal to, or greater than <code>src</code>

Table 7: Commonly used x86-64 instructions

This way of writing assembly code, presented in Table 7, is called AT&T syntax. Another syntax commonly used to read disassembler output is called Intel syntax. In the scope of our project, Intel syntax differs from AT&T syntax in that, the *dest* register (i.e., the register which will be modified) is the first operand of the instruction instead of being the second one.

This project uses a disassembler called *objdump*[10] through the examples presented. A disassembler is an application software responsible to convert a program from its binary representation to its assembly representation.

1.4.2 Overview of the Calling Conventions of System V ABI

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nevertheless, the ABI [11] recommends that all functions use the standard calling sequence when possible.

From Intel’s documentation [d1], “The System V Application Binary Interface defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement the interfaces defined in the X/Open Common Application Environment Specification, Issue 4.2 (also known as the “Single UNIX Specification”) and the System V Interface Definition, Issue 4. This includes, but is not limited to, systems that have implemented UNIX System V, Release 4.”

The System V Application Binary Interface (ABI) defines the calling conventions for how functions in a program interact at the binary level, specifically focusing on how arguments are passed, how the call and return instructions operate, and how the stack is managed.

One of the primary purposes of the System V ABI's calling conventions is to establish a standardized method for argument passing between functions. In this convention, the first six integer or pointer arguments are passed using specific registers (RDI, RSI, RDX, RCX, R8, and R9 on x86-64 architecture). If there are more than six arguments, the additional ones are passed on the stack. Arguments of type floating point, are passed using the registers XMM0 to XMM7.

This method minimizes memory access and leverages fast register operations, improving performance.

The call instruction is used to transfer control to a function. When a function is called, the return address (the address of the instruction immediately following the call) is pushed onto the stack. This ensures that the CPU knows where to return once the function execution is complete. The ret instruction, which stands for return, is used at the end of the function to pop the return address from the stack and jump back to that location, resuming execution of the caller function.

The stack plays a crucial role in function calls, particularly in managing local variables, passing additional arguments, and saving the state of registers. When a function is called, a new stack frame is created. This frame includes space for the return address, the function's local variables, and the saved registers that need to be restored when the function returns.

Callee-save and caller-save conventions are essential for preserving register values across function calls. Callee-save registers (such as RBX, RBP, and R12-R15 on x86-64) must be preserved by the called function. This means that if the callee modifies these registers, it must save their original values at the start of the function and restore them before returning. On the other hand, caller-save registers (such as RAX, RCX, RDX, and R8-R11) are the responsibility of the calling function. If the caller needs to preserve the values in these registers across a function call, it must save them before calling the function and restore them afterward.

By adhering to these conventions, the System V ABI ensures consistency and predictability in how functions interact, enabling code compiled from different sources or written in different languages to work together seamlessly. This

standardization is vital for system-level programming, compiler writers for example are inspired by those standards.

1.4.2.1 *Caller save, callee save, and return value*

The ABI defines how registers keep or lose their values across calls, we label the function invoking other functions as a caller and the called function as callee. Registers %rbp, %rbx, and %r12 through %r15 “belong” to the calling function, and

the called function is required to preserve their values. In other words, a called function must preserve these registers' values for its caller. This set of registers is also known as callee save.

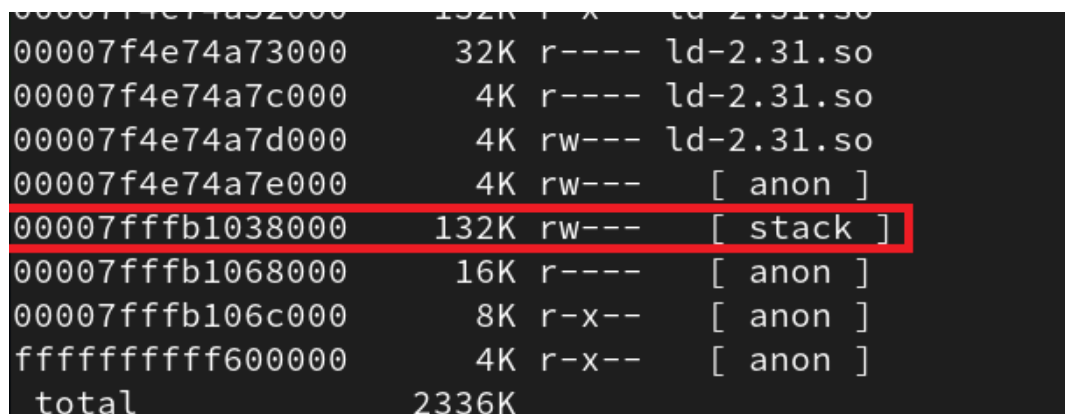
The remaining registers “belong” to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

When a function wants to return a value of type integer or address to a memory location, this value is placed in the register RAX.

When a function wants to return a floating-point value, the value is placed in XMM0 register.

1.4.3 The Stack Evolution During Function Calls

The stack is a segment of memory used to store objects with automatic lifetime. Typical stack addresses on x86-64 look like 0x7ffd'9f10'4f58—that is, close to 2⁴⁷. The Figure 8.a shows a typical memory allocation of the stack region. The allocation is done by the kernel of the operating system.



00007f4e74a73000	32K	r----	ld-2.31.so
00007f4e74a7c000	4K	r----	ld-2.31.so
00007f4e74a7d000	4K	rw---	ld-2.31.so
00007f4e74a7e000	4K	rw---	[anon]
00007fffb1038000	132K	rw---	[stack]
00007fffb1068000	16K	r----	[anon]
00007fffb106c000	8K	r-x--	[anon]
fffffffffff600000	4K	r-x--	[anon]
total	2336K		

Figure 8.a: Stack allocation in memory

The stack is named after a data structure, which was sort of named after pancakes[12]. Stack data structures support at least three operations: push adds a new element to the “top” of the stack; pop removes the top element, showing whatever was underneath; and top accesses the top element (see section 1.4.1.2). Note what’s missing: the data structure does not allow access to elements other than the top. (Which is sort of how stacks of pancakes work.) This restriction can speed up stack implementations.

Like a stack data structure, the stack memory segment is only accessed from the top. The currently running function accesses its local variables; the function’s caller, grand-caller, great-grand-caller, and so forth are dormant until the currently running function returns.

A function stacks look like this:

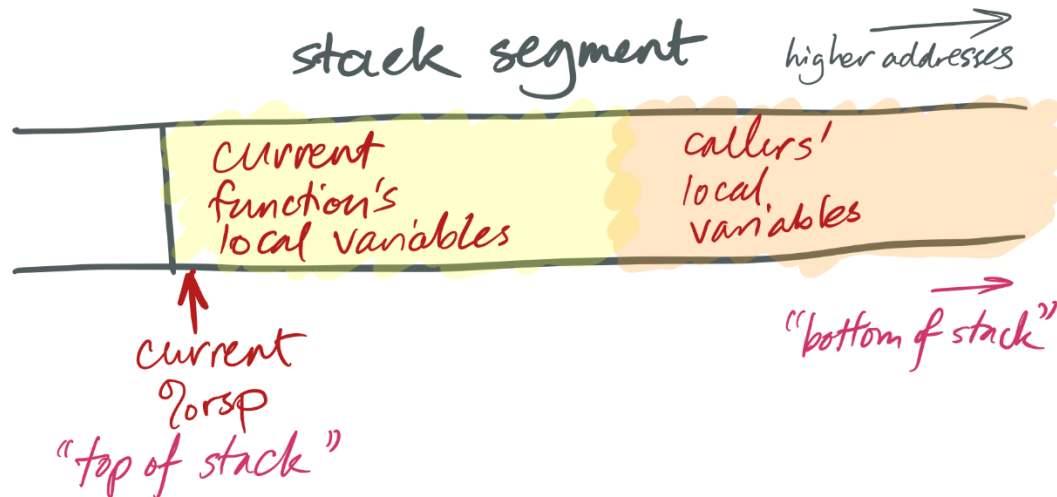


Figure 8: Multiple stack frames, each per function

The x86-64 `%rsp` register is a special-purpose register that defines the current “stack pointer.” This holds the address of the current top of the stack. On x86-64, as on many architectures, stacks grow down: a “push” operation adds space for more automatic-lifetime objects by moving the stack pointer left, to a numerically smaller address, and a “pop” operation recycles space by moving the stack pointer right, to a numerically-larger address. This means that considered numerically, the “top” of the stack has a smaller address than the “bottom.”

Operation of instructions like `pushq`, `popq`, `call`, and `ret` directly impact the stack memory. A push instruction pushes a value onto the stack. This both modifies the stack pointer (making it smaller) and modifies the stack segment (by moving data there). For example, the instruction `pushq X` (see section 1.4.1.2, for information on opcodes) means:

```
subq $8, %rsp
movq X, (%rsp)
```

And `popq X` undoes the effect of `pushq X`. It means:

```
movq (%rsp), X
addq $8, %rsp
```

`X` can be a register or a memory reference. `X` is a destination operand in the case of a pop operation and a source operand in the case of a push operation.

The portion of the stack reserved for a function is called: the function's stack frame. Stack frames are aligned: x86-64 requires that each stack frame be a multiple of 16 bytes, and when a `callq` instruction begins execution, the `%rsp` register must be 16-byte aligned.

To prepare for a function call, the caller performs the following tasks in its entry sequence:

- The caller stores the first six arguments in the corresponding registers.
- If the callee takes more than six arguments, or if some of its arguments are large, the caller must store the surplus arguments on its stack frame. It stores these in increasing order, so that the 7th argument has a smaller address than the 8th argument, and so forth. The 7th argument must be stored at `(%rsp)` (that is, the top of the stack) when the caller executes its `callq` instruction.
- The caller saves any caller-saved registers (see the previous section).
- The caller executes `callq` instruction (see the overview on x86-64 machine architecture). This has an effect like `pushq $NEXT_INSTRUCTION; jmp FUNCTION` (or, equivalently, `subq $8, %rsp; movq $NEXT_INSTRUCTION, (%rsp); jmp FUNCTION`), where `NEXT_INSTRUCTION` is the address of the instruction immediately following `callq`.

This leaves a stack like this:

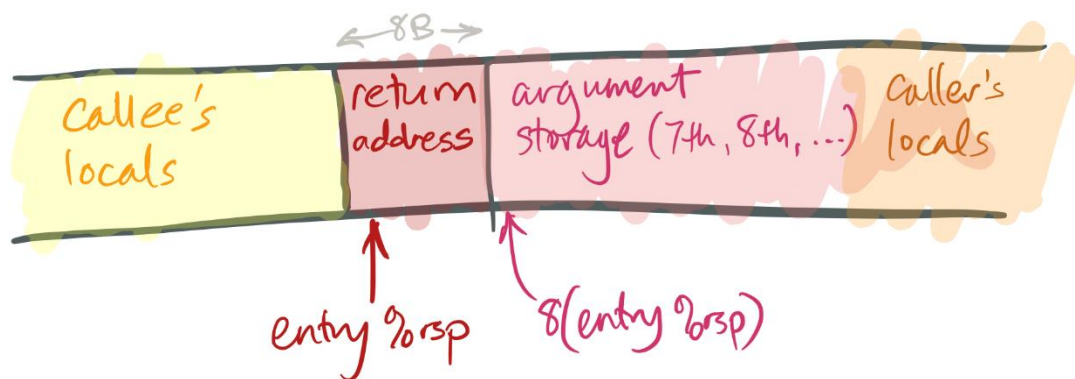


Figure 9: the stack after the call

To return from a function:

- The callee places its return value in `%rax`, if it is of type integer, or in `%MMX0` for a floating-point return value.
- The callee restores the stack pointer to its value at entry ("entry `%rsp`"), if necessary.
- The callee executes the `retq` instruction. This has an effect like `popq %rip`, which removes the return address from the stack and jumps to that address.
- The caller then cleans up any space it prepared for arguments and restores caller-saved registers if necessary.

Particularly simple callees don't need to do much more than return, but most callees will perform more tasks, such as allocating space for local variables and calling functions themselves.

1.4.4 Concrete Example

```
int f(){  
    int a=1;  
    g(a);  
}
```

```
int g(int number){  
    number++;  
    return number;  
}
```

```
int main(){  
    return f();  
}
```

Figure 10: Function f calls function g

Figure 10 shows a simple C program, it defines two functions, f and g, the function f calls the functions g, g executes and returns to f, then f will return the same value returned by g.

The function main is the execution starting point of any C program, in this case, it calls the function f without arguments, as control passes to the function f, the function f calls the function g with the argument a=1. When control goes back to f, f returns the return value of the function g. The function g takes one argument and it returns the argument incremented by one.

The program is compiled using gcc which is a compiler driver, it drives the process of compilation and linking of the program (see section 1.3). When executed, the program returned 2.

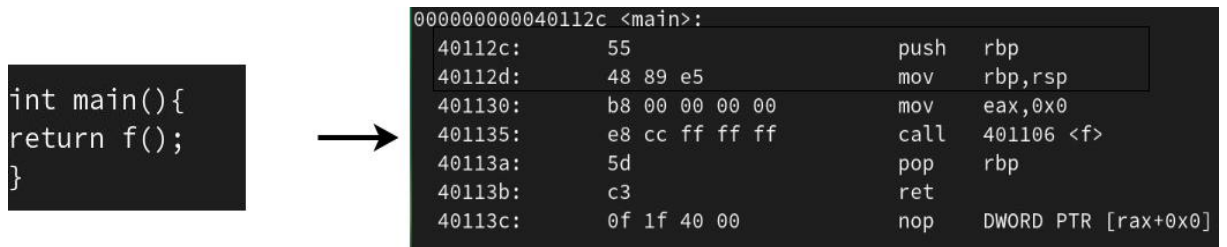


Figure 11: The main function in assembly representation

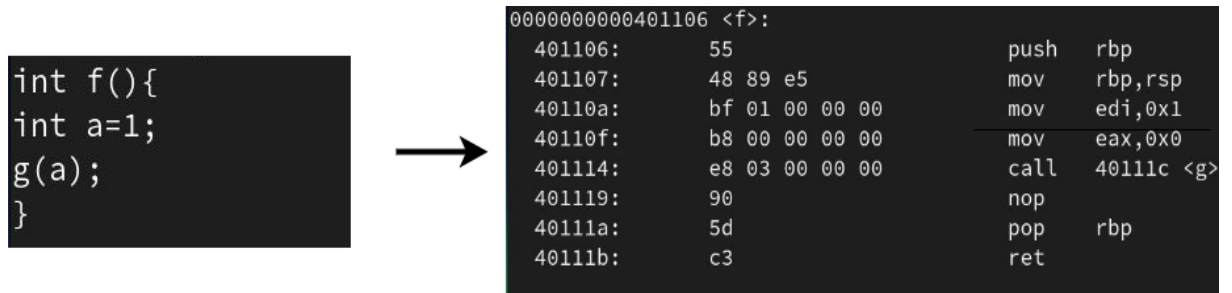


Figure 12: Function f in assembly representation

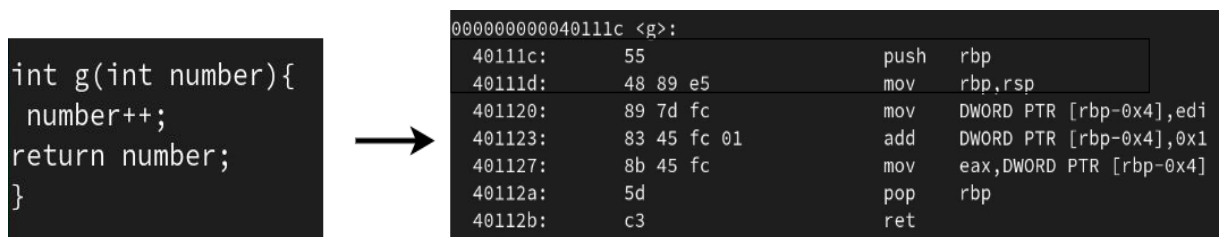


Figure 13: Function g in assembly representation

As figures 11,12 and 13 show, the function main is mapped to address (40112c)_{hex}, f to address (401106)_{hex}, and the function g is mapped to address (40111c)_{hex}.

I used gdb debugger to track the execution of this program, the results are shown below, and they seem compliant with the system V ABI definition of the calling conventions:

Before calling a function that accepts arguments, the caller must store the argument(s)'s values in the corresponding registers.

Throughout the documentation, figures are used to facilitate the understanding of some details during the execution of a C program in a computer system. Those figures are sometimes put in the chronological order of the instructions being executed.

The figures reduce the computer system as a memory holding code and data for a C program, connected to the MMU, the memory management unit, a hardware component discussed in section 2.2.3.1, a CPU, and a set of registers.

When a CPU cycle starts, the address contained in the RIP register is translated to a physical address, this is called an instruction fetch. Fetching an instruction requires a memory read operation, the instruction saved at the physical address provided by the MMU will be copied to a register illustrated as IR (instruction register). Whenever an instruction is fetched the RIP register is incremented by the size of the instruction in bytes, pointing to the next instruction.

The second phase of the cycle of the CPU is decoding, the CPU will decode the instruction, and changes its state before executing it. This phase also requires fetching the operands of the instruction being decoded from memory or registers.

The decoding phase is followed by the execution of the operation addressed by the instruction, and potentially by the saving of any value back to a memory location or a register.

After checking for interrupts the CPU will start the cycle again by fetching the next instruction. Interrupts are out of the scope of this project. They can be imagined like events that stop the execution of the program permanently or for a period of time.

The figures also show, where the RSP and RBP registers are pointing to, and what instruction is executed.

Figure 14 illustrates the argument passing mechanism, before the function *f* calls the function *g*, it must prepare arguments for it, in this case only one argument of value one gets passed within the register RDI.

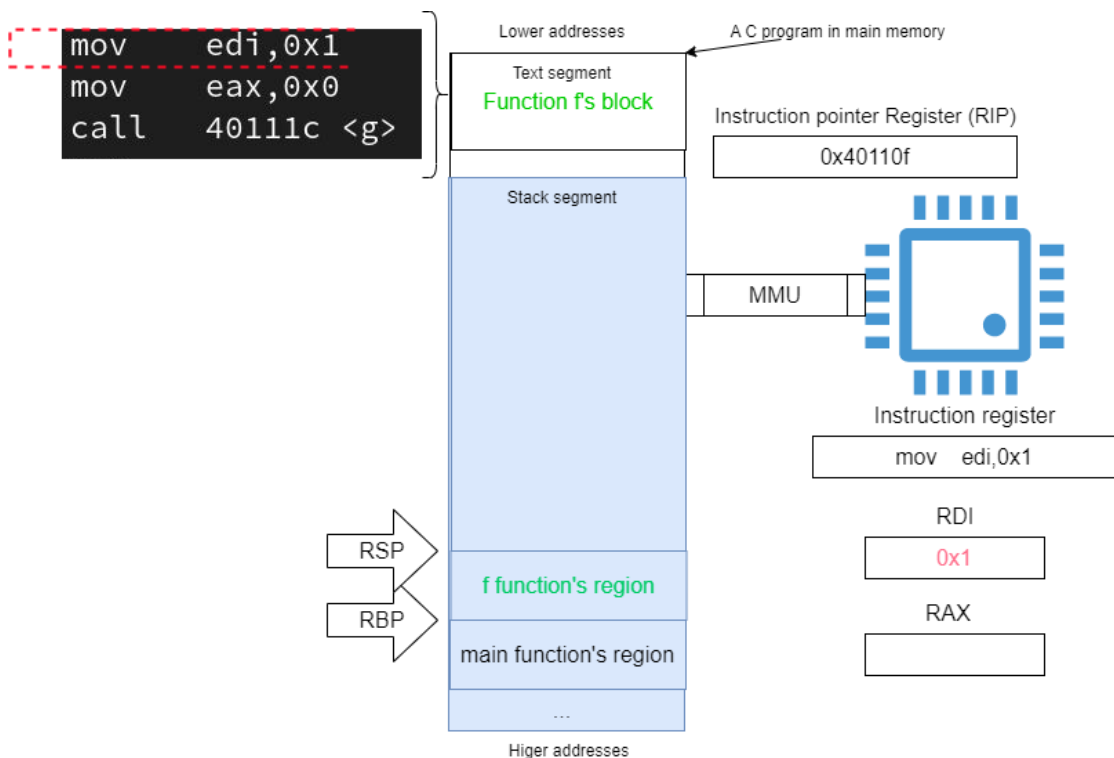


Figure 14: Passing one argument

Call <40111c> at address (401114)_{hex} is instructing the CPU to copy the address (40111c)_{hex}, the address of the next instruction in the block of the function f, to the top of the stack (e.g. to the memory location pointed to by RSP register).

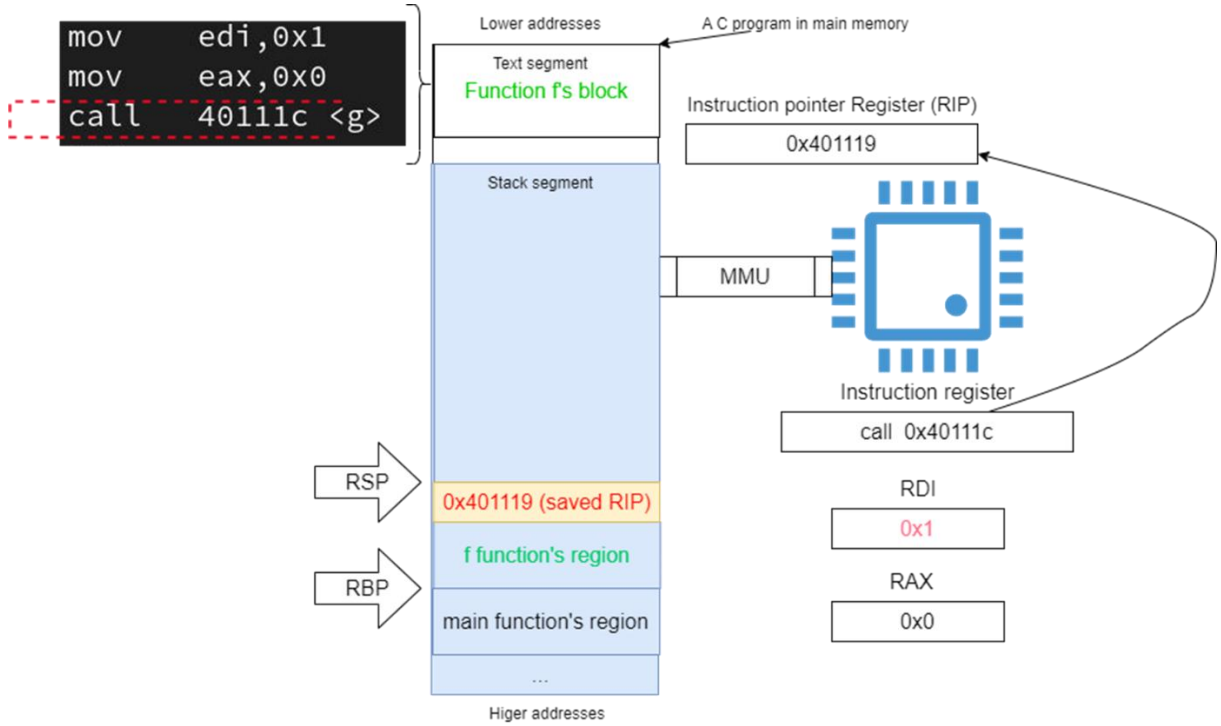


Figure 15: The call instruction

Then, the operand of the call instruction was copied to the RIP register, this will pass control to the function g at the next cycle of the CPU (see Figure 15).

The first two instructions in each function are identical, this is called a function prologue, a function prologue sets the stage for a new activation record, and it manipulates the value of RBP register to indicate a new stack frame.

push rbp instruction (e.g., the first instruction of the function prologue) moves the value of rbp register which holds the base of the stack of the caller (e.g., the address at which the activation record of the caller starts) to the stack, this is a callee save as defined in the previous section about System V ABI.

Push rbp instruction at address (40111c)_{hex} at the block of the function g means decrement rsp register by 8, the size of the content of RBP in bytes, and copy the value in rbp register at the location pointed to by rsp. So, the value of rbp will be stored at the next word after the return address towards lower addresses on the stack (see Figure 16).

The second instruction in the function prologue, *mov rbp, rsp*, is instructing the processor to copy the value in rsp register to the rbp register. This means that the callee (e.g., the called function) is setting a new activation record on the stack, so

when it calculates addresses for its local variables, the called function uses the new rbp as a base register.

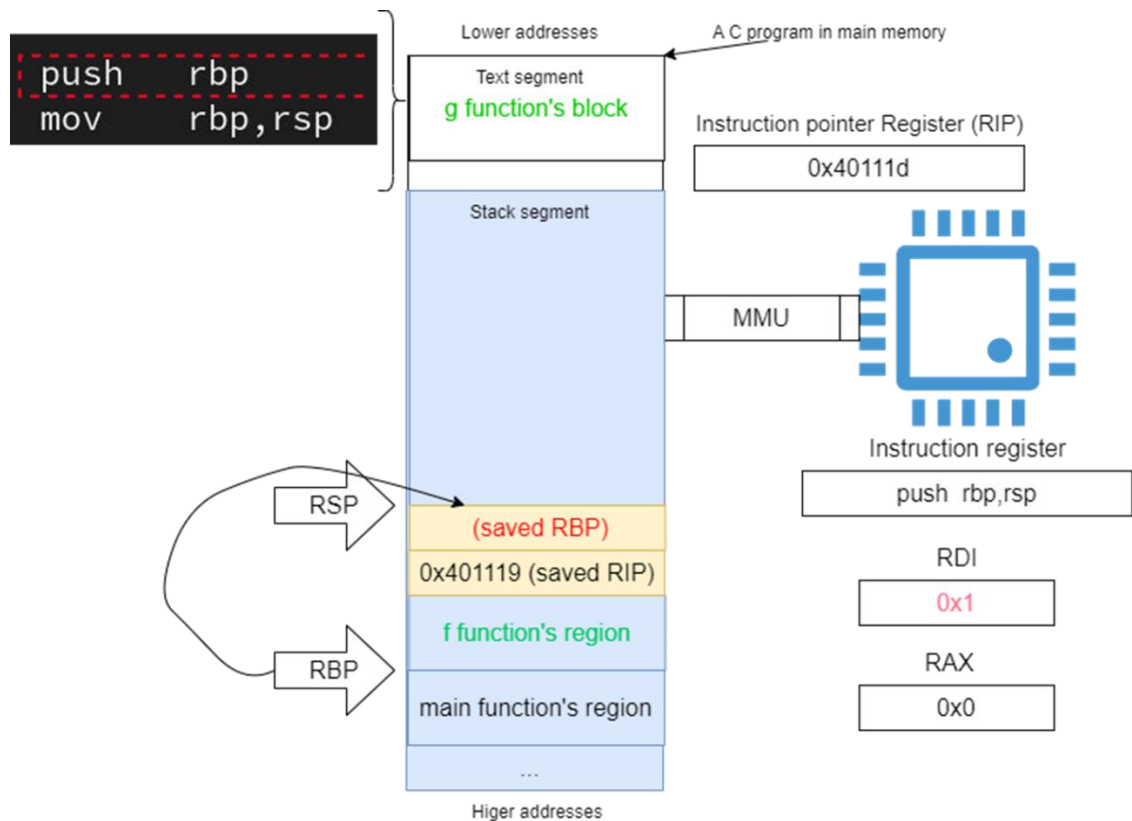


Figure 16: RBP register saved by the callee

The second instruction in each function's prologue is manipulating the value of RBP, making this register point to the memory location where the saved RBP is stored (see Figure 17).

This new value of RBP will be used by the callee (i.e., the function g) whenever it calculates an address of its local variables. This register indicates the stack frame of each function.

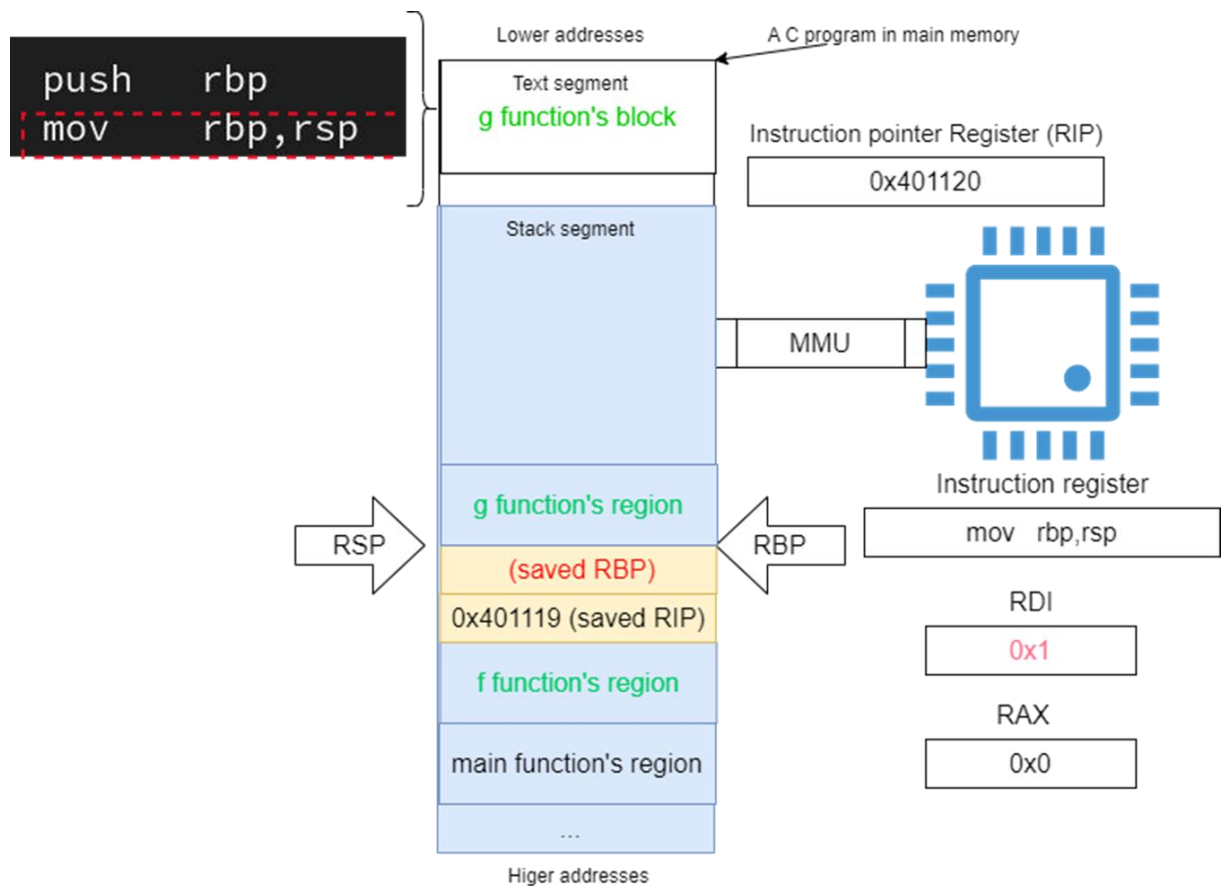


Figure 17: Indicating a new stack frame

When `g` finishes its calculation, two instructions (the instructions at addresses 40112a and 40112b), forming what is known as a function epilogue, would be executed.

The function `g` must return a value, before passing control back to `f`. In this case, only one integer return-value is passed using `rax` register (see Figure 18).

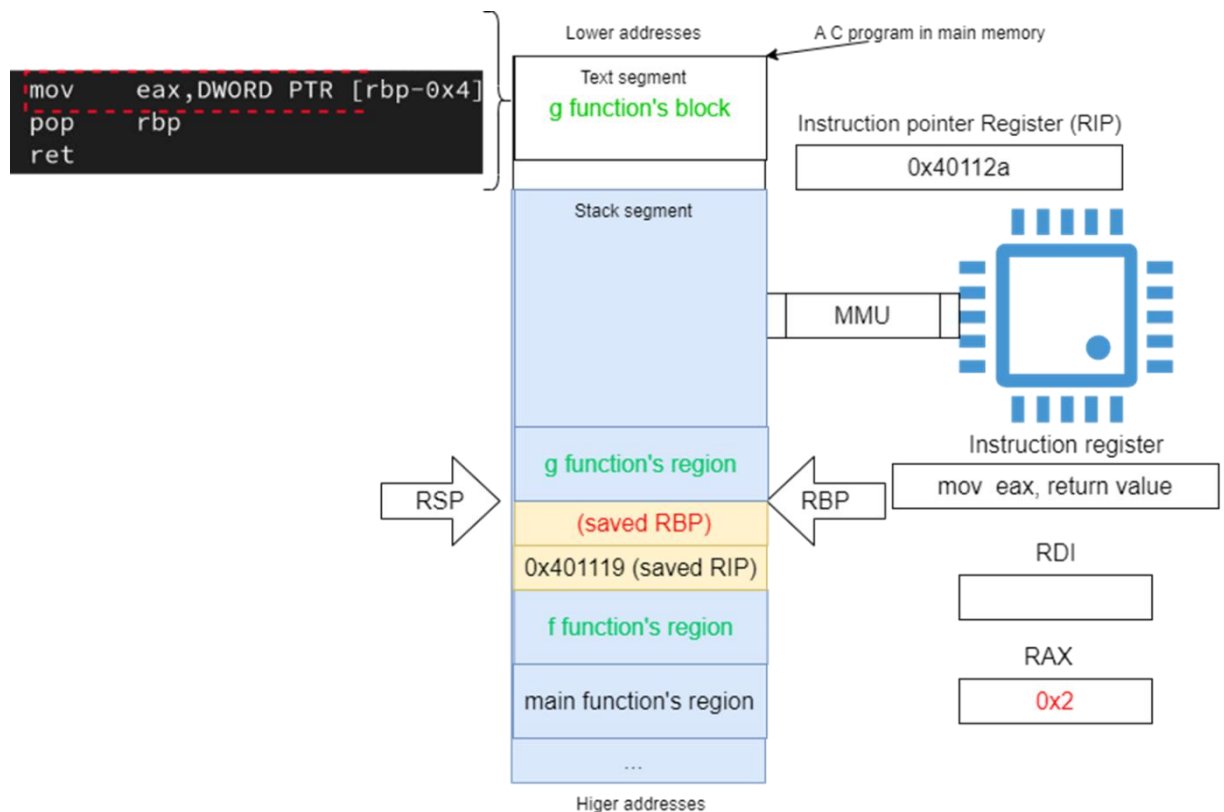


Figure 18: Passing integer return value

When the rip reaches the `pop %rbp` instruction inside the block of function g, `rsp` at this point is referring to the location where the base of the stack used by f is saved. The instruction `pop %rbp`, when executed, will restore the value of `rbp` as it was before the call to g (see Figure 19.)

After the `pop` instruction is executed `rsp` is incremented by the number of bytes an x86-64 address takes (e.g., 8 bytes).

This pattern of saving the address of the next instruction after the call instruction and saving the value of `rbp` register at the address referred by `rsp`, in this order, is repeated also when main calls f.

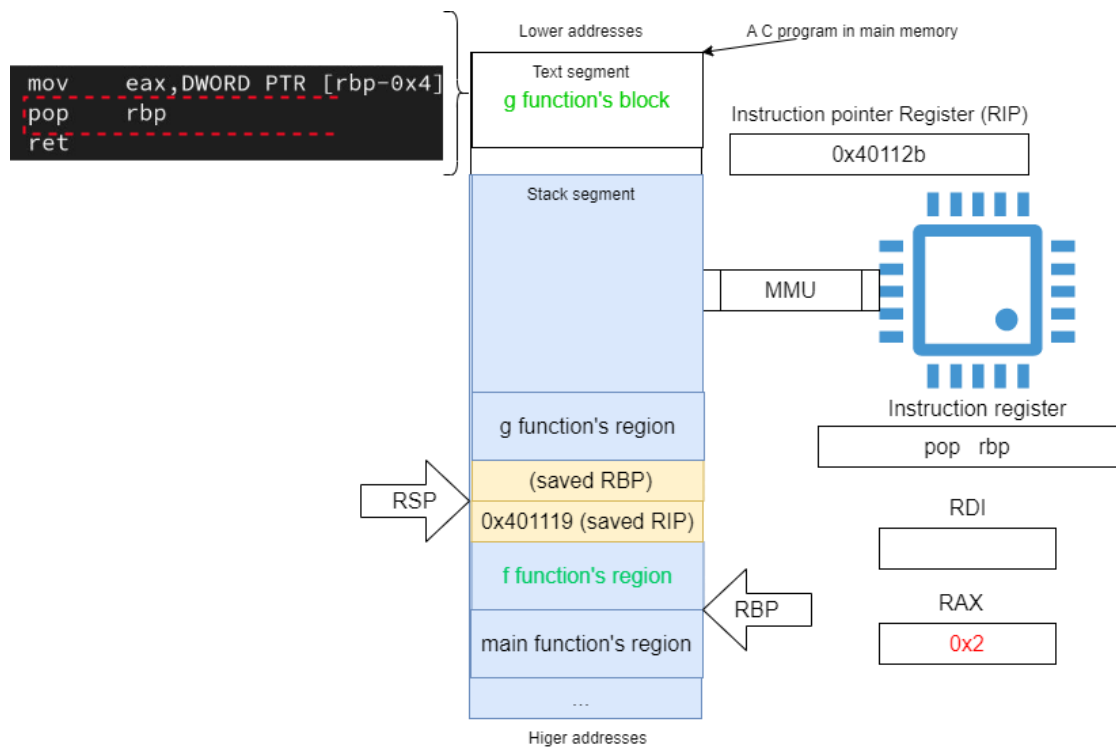


Figure 19: restoring the value of RBP

The second instruction in a function epilogue is the `ret` instruction. This instruction causes the CPU to pop the return address (i.e., the saved value of RIP register) from the stack into the RIP register, thereby transferring control back to the caller function. Upon execution, the stack pointer (RSP) is incremented to remove the return address from the stack, effectively cleaning up the stack frame of the called function (see Figure 20).

Figure 21, shows how the stack is left after returning to the caller.

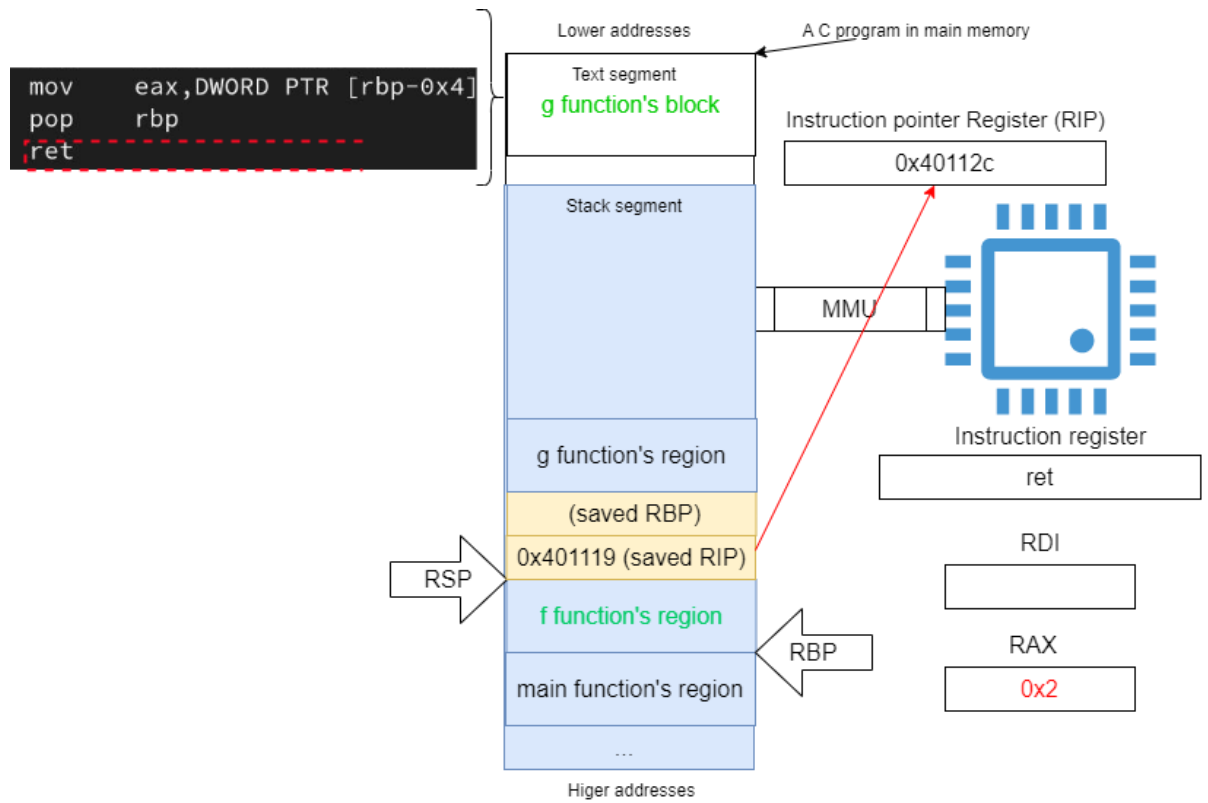


Figure 20: The ret instruction

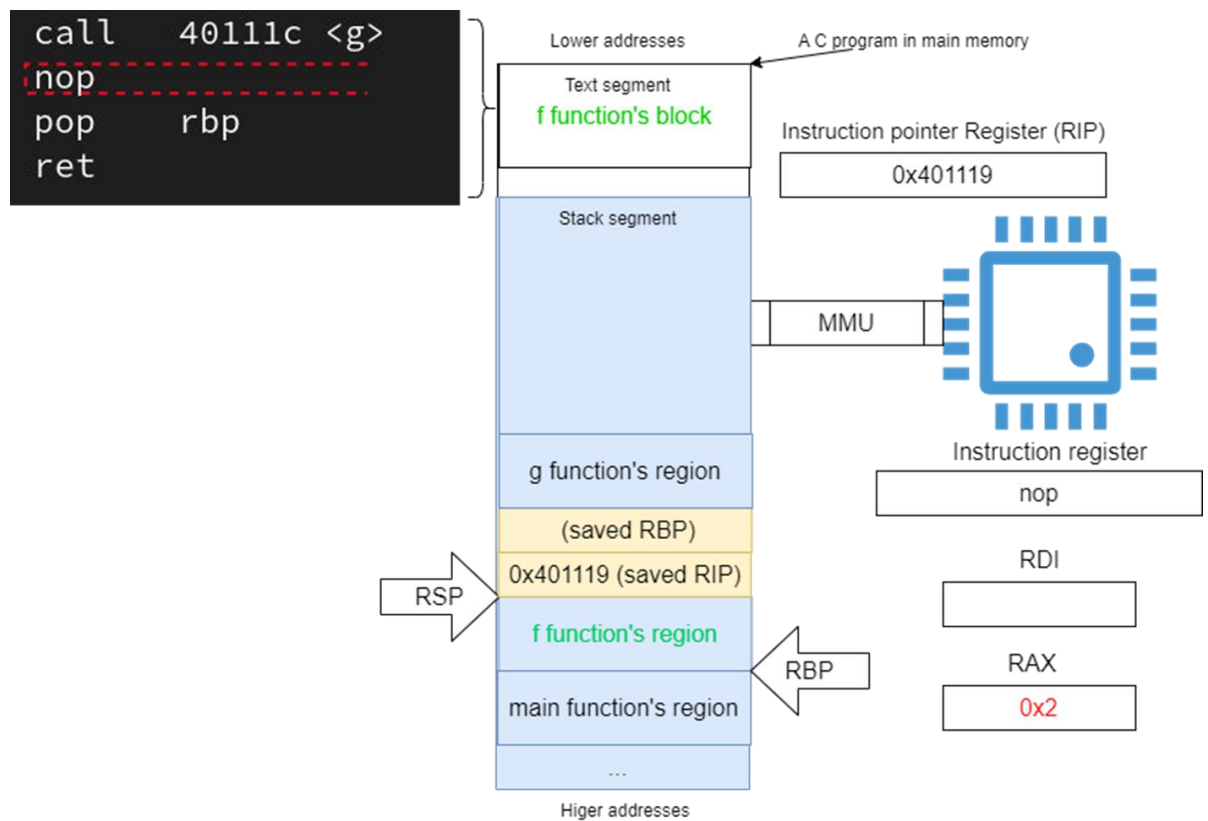


Figure 21: The end of the call sequence

1.5 Conclusion

Our inspections on the intervention of the stack memory in function calls show that two kinds of data are stored there. The saved instruction pointer (saved RIP) and normal data such as local variables are mixed together. Critical control flow data presence along with user defined data is putting the control flow of the program into risk, if the user defined data isn't handled carefully. This design is the cause of stack-based buffer overflow, and it will be clearer how this design leads to exploitable bug in the following sections where we focus on the bug on more details.

Chapter 2 Functions Lacking Bounds Checking

2.1 Introduction

In computer programming, bounds checking [13] is any method of detecting whether a variable is within some bounds before it is used. It is usually used to ensure that a number fits into a given type (range checking), or that a variable being used as an array index is within the bounds of the array (index checking). A failed bounds check usually results in the generation of some sort of exception signal.

As performing bounds checking during each use can be time-consuming, it is not always done. Bounds-checking elimination is a compiler optimization technique that eliminates unneeded bounds checking.

This chapter addresses the critical issue of bounds checking in C programming. We explore the risks posed by buffer overflows due to the language's lack of built-in protections, including common vulnerabilities like those found in functions such as `strcpy()`. We'll examine how lack of bounds checking is exploited by attackers, existing mitigation techniques, and a proposed solution by statically substituting this function with its relative `strncpy()` without the need for the source code.

A contiguous set of memory locations is known as a buffer. When a function wants to deal with such a buffer, a pointer to that memory location is passed as an argument to that function. A lot of commonly used functions, provided by the C standard library, don't implement a way to check the size of the buffers passed as arguments. Thus, if there is a way to control the size of the buffer other regions in memory, not reserved for the argument, would be corrupted, leading to a potential binary exploitation.

2.2 GNU C Library

The vulnerable function provided as an example in this work is `strcpy()` is part of the C standard library. The GNU C library [14] is a specific implementation of the C standard library. Commonly abbreviated as `glibc`, is a fundamental component of most Unix-like operating systems. It is a core part of the GNU project and serves as the standard C library for these systems. Developed by the Free Software Foundation (FSF), `glibc` provides essential functionality to programs written in the C programming language, including input/output operations, memory allocation, and system calls.

At its core, `glibc` is designed to be highly portable, supporting a wide range of hardware architectures and operating system kernels. This portability allows software developers to write code that can run on various Unix-like systems without

modification. One of the key features of glibc is its adherence to various standards, particularly the ISO C and POSIX standards. By conforming to these standards, glibc ensures that programs written against its APIs behave predictably across different platforms. This adherence also facilitates interoperability between different software components and systems. In addition to standard C library functions, glibc includes extensions and optimizations to improve performance and functionality. These extensions cover areas such as internationalization, threading, and networking, enhancing the capabilities of applications running on Unix-like systems.

Overall, glibc plays a crucial role in the functioning of Unix-like operating systems, providing a robust and standardized foundation for software development and system operations. Its portability, standards compliance, and ongoing development make it an indispensable component of the open-source ecosystem.

2.3 Unsafe Functions: an Example of `strcpy()`

Glibc, like many software libraries, includes functions that lack bounds checking, which can lead to vulnerabilities if not used carefully. To name a few, the following list (Table 22) includes some popular glibc functions known to lack bounds checking:

<code>strcpy</code> : Copies a null-terminated string from the source to the destination buffer without performing bounds checking.
<code>strcat</code> : Appends the source null-terminated string to the end of the destination null-terminated string, also without bounds checking.
<code>gets</code> : Reads a line from standard input into a buffer until a newline or EOF is encountered, but lacks bounds checking and is considered unsafe.
<code>sprintf</code> : Formats and stores a series of characters and values into a buffer, similar to <code>printf</code> .
<code>scanf</code> : Reads formatted input from standard input, parsing it according to the provided format string.
<code>realpath</code> : Resolves a relative path to an absolute path and stores it in a buffer, which must be sufficiently large.
<code>memcpy</code> : Copies a specified number of bytes from a source memory location to a destination.
<code>strtok</code> : Tokenizes a string, splitting it into substrings based on specified delimiter characters, modifying the original string in the process.

Table 22: Popular vulnerable functions

Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account. The `strcpy()` function, used as an example in this work, is unsafe

because it does not perform bounds checking on its arguments. When used uncarefully, it can lead to out-of-bounds memory writing, corrupting memory locations that aren't concerned by the function of `strcpy`.

`strcpy()` is used to copy, byte by byte, a string of characters pointed to by its second argument, to a buffer pointed to by the function's first argument until a null terminator is encountered. However, it does not perform any bounds checking, meaning it will continue copying characters from the source string until it reaches a null terminator, potentially overflowing the destination buffer if it is not large enough to hold the entire string. This can result in buffer overflow vulnerabilities, a common security issue where an attacker can exploit the lack of bounds checking to overwrite adjacent memory locations with malicious code or data.

For example, consider the following code snippet:

```
char dest[10];  
  
char source[] = "This is a long string";  
  
strcpy(dest, source);
```

In this case, `strcpy()` will attempt to copy the entire contents of the source array (including the null terminator) into the dest array, which only has space for 10 characters. As a result, it will overflow the dest buffer, leading to undefined behavior and potentially exploitable security vulnerabilities.

To mitigate the risk of buffer overflow vulnerabilities, developers should use safer alternatives that perform bounds checking, such as `strncpy()`. `strncpy` function allows developers to specify the maximum number of characters to copy, preventing buffer overflows if the source string is longer than the specified number.

2.3.1 The Bug in Detail

A buffer is a contiguous region of memory, it is a set of adjacent memory locations. An example of a buffer is an array of any type. The buffer is identified by the memory address of its first element.

In our explained example, the buffer is a null-terminated string of characters. It can be the first argument of the `strcpy` function (the destination buffer), as it can be the source buffer (the second argument of the function).

Generally, the problem occurs when a programmer uses a function that copies a chunk of memory from one buffer to another without taking the size of each buffer into account. For reasons of simplicity, this work takes the example of `strcpy()`. The function takes two arguments, the first argument (the destination argument) is a pointer to the destination buffer, on which the string will be copied,

and the second argument (the source argument) is an address pointing to where the string is stored in memory.

In some cases, the programmer is giving the user of the program the ability to store a string of an arbitrary size in the buffer pointed to by the source argument. In this case, if the destination buffer can't support the user's chosen size (e.g., the size of the destination buffer is smaller than the size of the source buffer), the function using (i.e., calling) `strcpy()` will have a corrupted stack frame. Especially, memory locations adjacent to the destination buffer will be corrupted.

As we explored in the previous chapter, data that contributes to the flow of control of a program is stored on the stack, this is the saved return address. The saved return address is pushed on the stack after a function executes a call instruction to another function, so, when the called function finishes its computation and executes the *RET* instruction, the execution will continue at the block of the caller, in the instruction following the call instruction. This data is targeted by attackers by exploiting stack overflow vulnerability. It is corrupted to manipulate the flow of the program that uses `strcpy` function or any similar function in terms of the discussed vulnerability.

When the saved return address gets corrupted, whenever a `ret` instruction is executed in the function that uses `strcpy`, control flow will be passed to the instruction pointed to by the corrupted data.

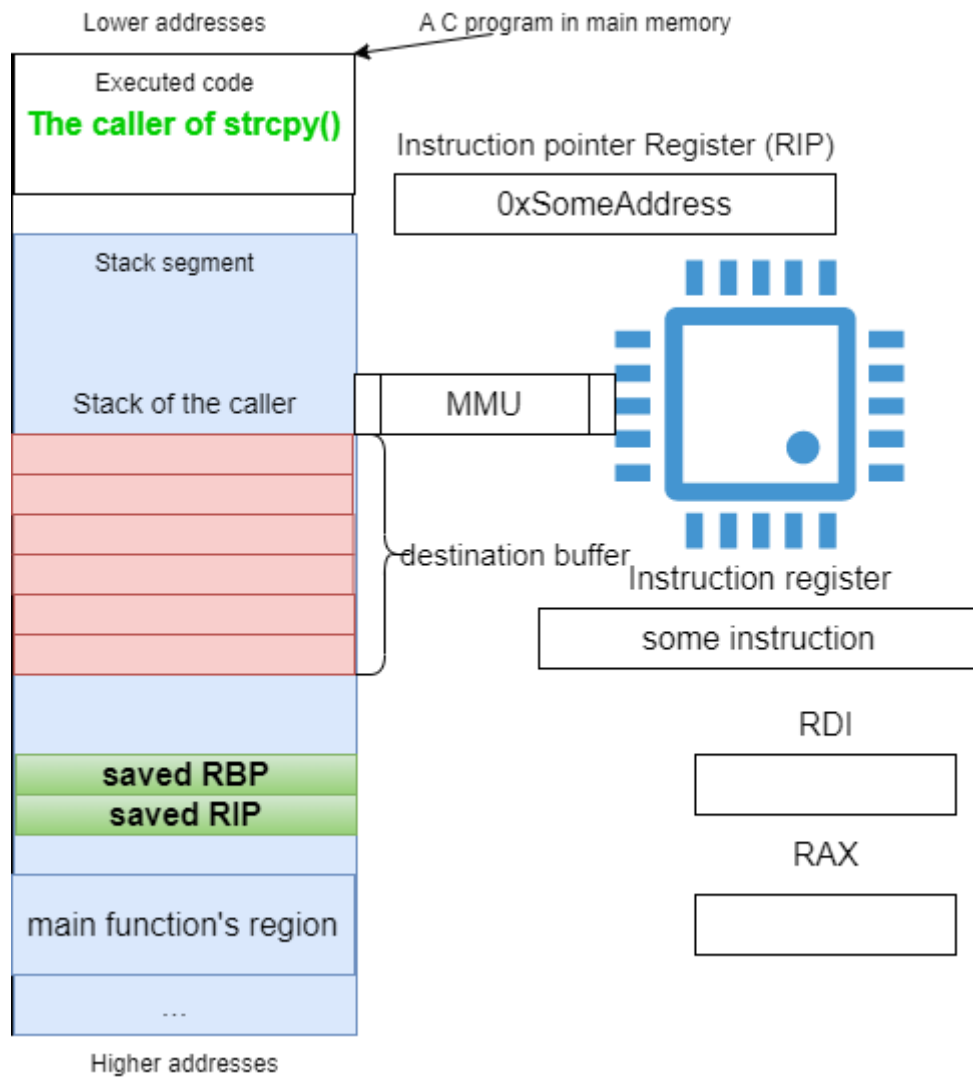


Figure 22: A program using strcpy

At compile-time, as Figure 22 shows, the compiler will reserve space for the destination buffer as dictated by the developer.

The uncareful use of `strcpy`, will overflow the destination buffer, leaving the stack somehow like what is shown in Figure 23.

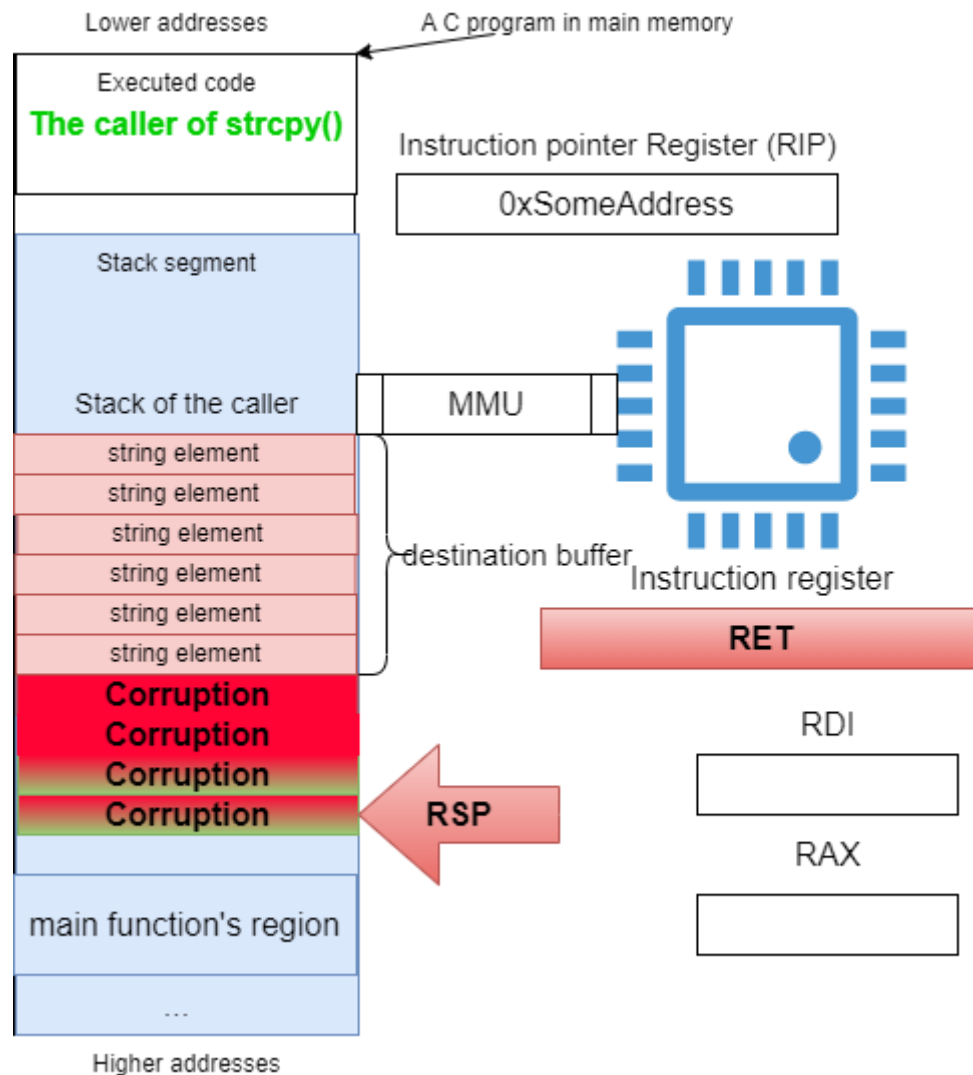


Figure 23: Corrupted caller's stack frame

2.3.1.1 *strcpy* function

From the Linux man page of this function, we see the description below (figure 24):

The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy. Beware of buffer overruns! (See BUGS.)

The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. Warning: If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated.

If the length of `src` is less than `n`, `strncpy()` writes additional null bytes to `dest` to ensure that a total of `n` bytes are written.

A simple implementation of `strncpy()` might be:

```
char *
strncpy(char *dest, const char *src, size_t n)
{
    size_t i;

    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';

    return dest;
}
```

This function replaces `strcpy()` and it checks the bounds of the source buffer. This function will be used in **section 2.3.1** where a mitigation by substitution is proposed.

Return Value

The `strcpy()` and `strncpy()` functions return a pointer to the destination string `dest`.

Bug

If the destination string of a `strcpy()` is not large enough, then anything might happen. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there's enough space. This may be unnecessary if you can show that overflow is impossible, but be careful: programs can get changed over time, in ways that may make the impossible possible.

Figure 24: Linux man page of `strcpy`

Consider the source code of a program using the flawed `strcpy` function (Figure 25). The developer of this program neglected the situation where the user can provide a string of more than 15 characters.

```
#include <stdio.h>

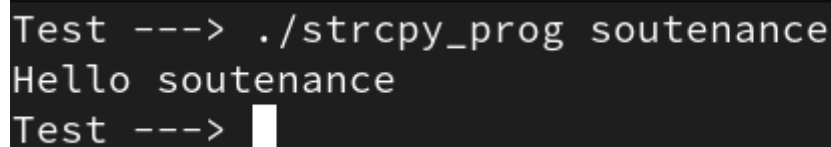
#include <string.h>
```



```
void printName(char* buffer) {  
    char name[16];  
    strcpy(name, buffer);  
    printf("Hello %s\n", name);  
}  
  
int main(int argc, char* argv[]) {  
    if(argc > 1) printName(argv[1]);  
    return 0;  
}
```

Figure 25: A program invoking strcpy

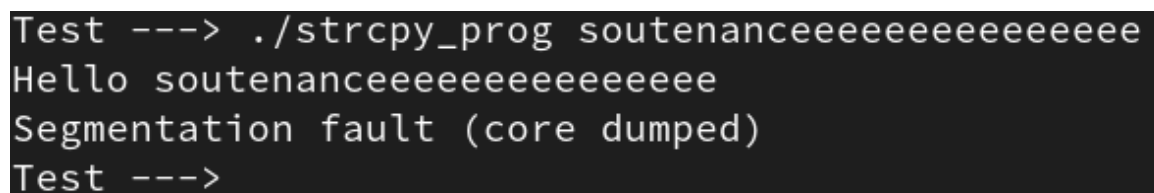
Executing the program with a legitimate input gave an expected behavior as in Figure 26.



```
Test ---> ./strcpy_prog soutenance  
Hello soutenance  
Test ---> 
```

Figure 26: Normal execution

Executing the program with an argument of more than 16 characters stops the program (see figure 27).



```
Test ---> ./strcpy_prog soutenanceeeeeeeeeeeeeeeeeee  
Hello soutenanceeeeeeeeeeeeeeeeeee  
Segmentation fault (core dumped)  
Test ---> 
```

Figure 27: An execution with unintended input

The program was stopped because the saved return address was corrupted by the ASCII value of the character 'e'.

During the execution of the program, at the point before the program was stopped. The inspection of the stack using gdb [15] debugger is shown in Figure 28.

```

Breakpoint 1, 0x0000000000401172 in main ()
(gdb) c
Continuing.
Hello soutenanceeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee

Breakpoint 2, 0x000000000040116d in printName ()
(gdb) x/1cg $rsp
0x7fffffffdd48: 101 'e'
(gdb) x/5cg $rsp
0x7fffffffdd48: 101 'e' 101 'e'
0x7fffffffdd58: 101 'e' 0 '\000'
0x7fffffffdd68: -126 '\202'
(gdb)

```

Figure 28: Debugger view of the program

We stopped the execution of the program before the `ret` instruction in the block of the `printName` function was executed for debugging purposes. At this point of execution, the `rsp` register was pointing to the saved return address. Inspecting what value is stored at the location pointed to by `rsp`, shows some instances of the letter 'e'. This means that upon the execution of the `ret` instruction in the block of `printName`, control will be passed to a corrupted return address, in this case, the address: `0x6565656565656565` (see Figure 29).

```

0x7fffffffdd48: 101 'e' 101 'e'
0x7fffffffdd58: 101 'e' 0 '\000'
0x7fffffffdd68: -126 '\202'
(gdb) x/1ag $rsp
0x7fffffffdd48: 0x6565656565656565

```

Figure 29: The corrupted return address

However, Figure 30 shows that the program was stopped by the operating system due to a mitigation discussed in section 2.2.3.

```

(gdb) x/1ag $rsp
0x7fffffffdd48: 0x6565656565656565
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x000000000040116d in printName ()

```

Figure 30: Segmentation Fault exception

2.3.2 Exploit: Arbitrary Code Execution

Stack buffer overflow bug can be exploited to corrupt program's data. Data corruption can impact the control flow of a program, if this data is used to do so.

In worst cases, the bug is exploited by attackers causing the vulnerable program to execute arbitrary, attacker-chosen code.

Arbitrary code execution exploitation works by putting a well-crafted sequence of characters in the unprotected source buffer on which the attacker is having control.

This sequence of characters is divided to two parts, the first part is the code in machine language that the attacker wants to execute. The second part is a memory address pointing to the first byte of the first part. The address must coincide, in position, with the saved return address on the stack (see figure 31).

The favorite code in an attacker point of view is one which spawns a shell, a command interpreter program.

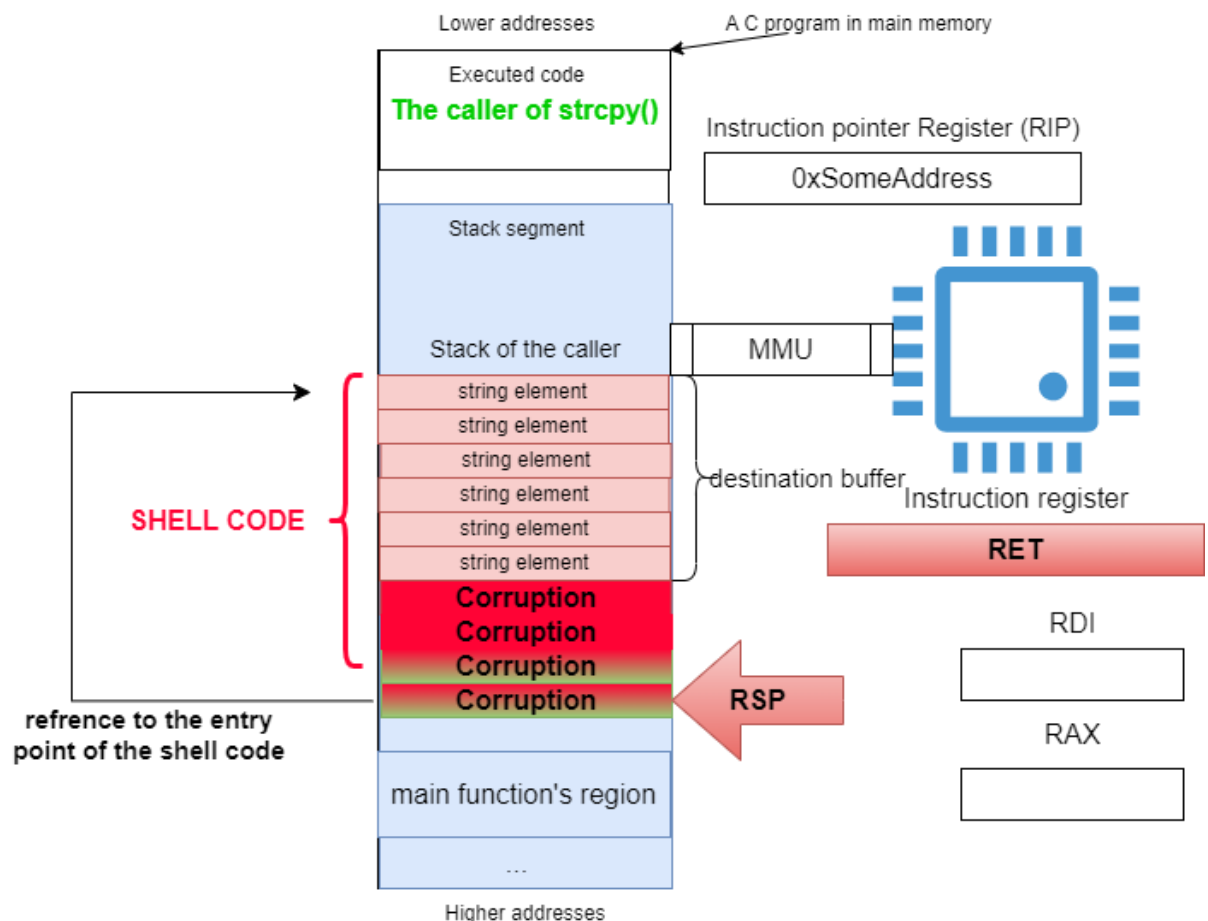


Figure 31: Inner workings of shellcode exploit

The shellcode must meet some constraints. It must avoid null bytes, as these are interpreted as string terminators and would truncate the shellcode when copied by functions like `strcpy`.

The size of the shellcode must be compact enough to fit into the buffer being overflowed. Also, Certain characters may need to be avoided depending on the context (e.g., newline characters, spaces, and others that could affect the copying process or execution flow).

2.3.3 Mitigation: access restriction to the stack

2.3.3.1 *The memory management unit*

This exploit is mitigated by designers using a special hardware component: MMU[16]. The Memory Management Unit is a hardware component lays between the CPU and the memory controller. It pertains to the CPU and it stores a data structure called page tables used to resolve virtual addresses to physical addresses. The MMU implements the concept of virtual memory, which allows a computer to appear to have more memory than it physically possesses. Each program running on the system operates within its own virtual address space, which is divided into fixed-size units called pages. When a program accesses memory, it uses virtual addresses. The MMU translates these virtual addresses into physical addresses, which correspond to specific locations in physical memory (RAM). This translation is performed using hardware-based memory management techniques. Each entry in the page table corresponds to a page of memory, containing the physical address where that page is stored.

If a program accesses a virtual address that is not currently mapped to a physical address, a page fault occurs. The MMU intercepts this fault and triggers a process called page fault handling. The operating system then determines the appropriate action, such as loading the required page from secondary storage (e.g., disk) into physical memory.

The MMU enforces memory protection by assigning access permissions to each page of memory. These permissions specify whether a page can be read from, written to, or executed. If a program attempts to access memory in violation of these permissions, the MMU raises an exception, typically resulting in a segmentation fault or similar error.

The mitigation is called NX (No-eXecute), it consists of restricting the stack to be accessed for code execution. Page tables are extended to specify the access rights for each region in memory. The stack region is accessed only for a read or a write operation. The NX bit works by marking memory pages with specific permissions that define whether code execution is allowed on those pages. In particular:

- Data Pages: Pages designated to store data (such as those used for the stack or heap) are marked as non-executable.
- Code Pages: Pages designated to store executable code (such as those containing the program's instructions) are marked as executable.

When the CPU encounters an instruction that attempts to execute code from a non-executable page, it triggers a hardware exception, preventing the execution of the

injected code. This mechanism effectively stops many common exploitation techniques, as it enforces a clear separation between executable code and data.

By implementing the NX bit, modern processors and operating systems can provide an additional layer of security, ensuring that only intended and authorized code is executed, thereby significantly mitigating the risk of arbitrary code execution through exploits like stack-based buffer overflows.

2.3.3.2 *Stack with read and write access permissions*

Designers thought that memory segments (set of pages) of the process which contain code must be marked as executable and read-only. On the other hand, those areas containing data are marked as read/write and non-executable. Processors must provide hardware support to check for this policy when fetching instructions from main memory. Even if an attacker successfully injects code into a writeable (not executable) memory region, any attempt to execute this code would lead to a process crash. This technique is also known as “W^X” because a memory page can be marked as executable or writable, but not both at the same time. Though this mechanism is implemented on the MMU, the operating system support is required:

At the startup time of a process, the operating system and the runtime loader use `mmap` extensively to set up the process's memory layout. This includes mapping the executable code, shared libraries, the stack, the heap, and other necessary regions into the process's address space. The usage of `mmap` at this stage is crucial for enforcing memory protections and supporting NX (No-eXecute) mitigation.

`mmap` is a system call, wrapped by a C function called also `mmap`[18]. In a typical unix-like system, `mmap` function is used by the loader at the startup time to allocate memory for the stack, it is invoked as follow:

```
mmap(stack_addr, stack_size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
```

The flags `PROT_READ` and `PROT_WRITE` specify the access permissions to the stack as read and write, this will configure the data structure stored in the MMU.

The Figure 32 highlights the allocation of the stack by tracing the use of system calls at the execution of a typical C program.

```
mmap(NULL, 1872744, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
= 0x7fe8ab179000
mmap(0x7fe8ab19e000, 1376256, PROT_READ|PROT_EXEC, MAP_PRIVATE|M
AP_FIXED|MAP_DENYWRITE, 3, 0x25000) = 0x7fe8ab19e000
mmap(0x7fe8ab2ee000, 307200, PROT_READ, MAP_PRIVATE|MAP_FIXED|MA
P_DENYWRITE, 3, 0x175000) = 0x7fe8ab2ee000
mmap(0x7fe8ab339000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MA
P_FIXED|MAP_DENYWRITE, 3, 0x1bf000) = 0x7fe8ab339000
mmap(0x7fe8ab33f000, 13160, PROT_READ|PROT_WRITE, MAP_PRIVATE|MA
P_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fe8ab33f000
close(3)                                = 0
```

Figure 32: Stack allocation using `mmap`

By visualizing the memory layout of a running process, we can also confirm the presence of this mitigation on modern systems (see Figure 33). The stack is mapped as readable and writable only.

00007f58ff272000	4K	r----	libnss_sss.so.2
00007f58ff273000	4K	rw---	libnss_sss.so.2
00007f58ff274000	28K	r--s-	gconv-modules.cache
00007f58ff27b000	8K	r----	ld-2.31.so
00007f58ff27d000	132K	r-x--	ld-2.31.so
00007f58ff29e000	32K	r----	ld-2.31.so
00007f58ff2a7000	4K	r----	ld-2.31.so
00007f58ff2a8000	4K	rw---	ld-2.31.so
00007f58ff2a9000	4K	rw---	[anon]
00007fffee905000	132K	rw---	[stack]
00007fffee944000	16K	r----	[anon]
00007fffee948000	8K	r-x--	[anon]
fffffffffff60000	4K	r-x--	[anon]

Figure 33: The memory layout of a running process

2.3.4 Evasion: Code Reuse Attack

The Non-eXecutable bit (NX)/Data Execution Prevention (DEP) mechanism can be bypassed using attacks that do not require to execute an injected code, but reuse the already existing and mapped code on the target application.

Code Reuse Attacks work by corrupting the return address on the stack to repurpose existing components that are mapped in the MMU as executable. In particular, they repurpose existing code to perform arbitrary computations (see Figure 34).

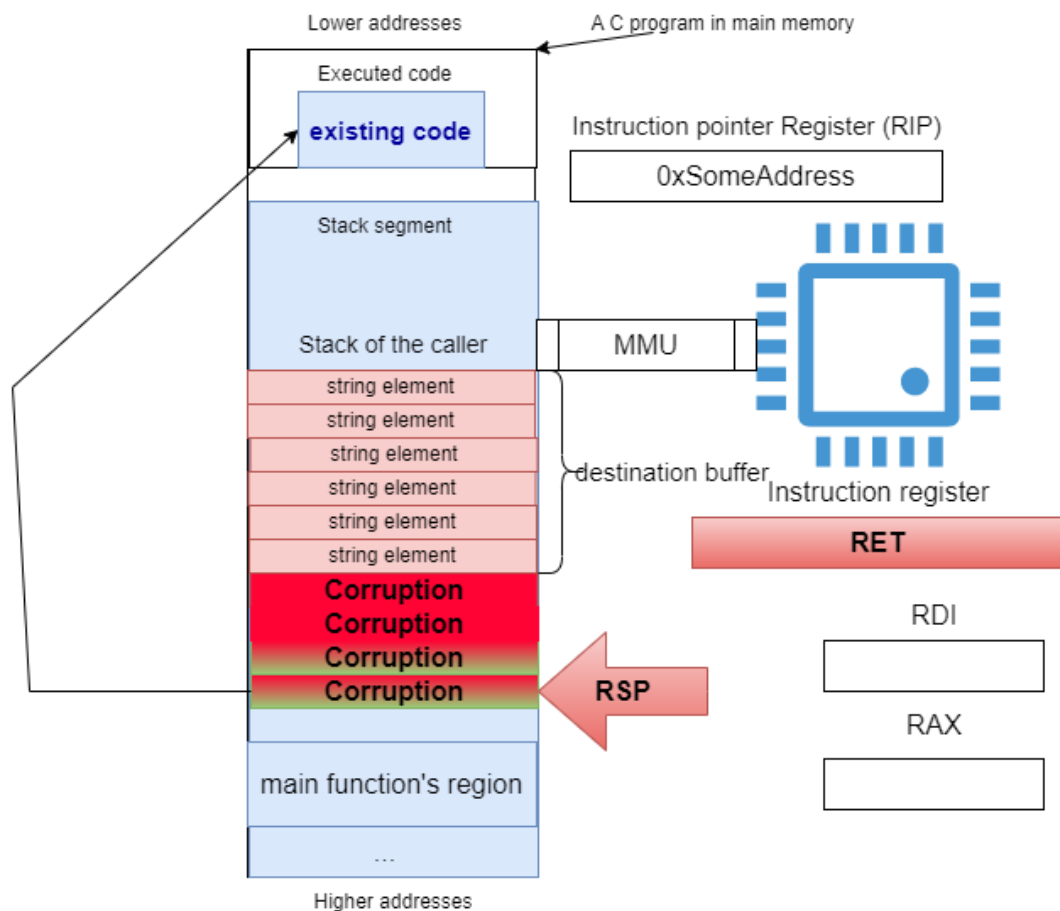


Figure 34: Code reuse attack

There is a family of techniques referred to as ret2* and more generally the Return Oriented Programming (ROP) technique. ROP is a very effective technique to bypass the NX mitigation. It is realistic to assume that modern attacks do not inject code but use the ROP method. Therefore, from now on we will assume that the NX bit protection is bypassed directly, and then the security relies on the effectiveness of the remaining security measures.

2.3.5 Mitigation: Stack Canaries

This section shows an additional mitigation that is done after programmers generate their source code, during compilation time. In the event that source code is absent, a method for achieving stronger protection is given in the next section. Stack Canaries are quite basic; we start the function with a random value added to the stack. The original random value is compared to the current value before the program performs ret; if they match, there hasn't been a buffer overflow. If they aren't, the software fails, frequently accompanied by a warning message stating that "stack smashing detected." The attacker then tries to overflow to take control of the saved return address.

There have been three different canary types proposed [17], each with advantages and disadvantages:

- Random: the canary is a random number, unknown to the attacker;
- Terminator: the canary contains characters that stop most string functions (newline, null byte, linefeed, -1);
- XOR: the canary is the XOR of a random value and the saved return address.

2.3.5.1 *Concept and Limitations*

If the attacker is unable to guess them, random canaries are a good idea. Sadly, memory leak issues have the ability to expose the value of the canary, making it completely worthless. Conversely, terminator canaries are constant values that the attacker already knows and can't modify. They utilize numerous strategies to thwart the attacks. Take a `strcpy()`-based overflow, for instance. Because `strcpy()` won't copy all the bytes that follow a terminator character, we know that the attacker's payload cannot contain a newline character. However, it will be discovered if the attacker replaces the newline in the canary with something different. Sadly there are flaws with `memcpy()`, `read()`, and even custom hand-written code because they do not depend on any special character.

Using a Random Canary or a Terminator Canary isn't efficient when the program has also vulnerabilities that allow for arbitrary memory write or arbitrary memory read. If an attacker can expose the random value of the Canary using a vulnerability like format string, he would easily corrupt the saved return address. Vulnerabilities that facilitate arbitrary memory write will allow the attacker to corrupt the saved return address without the need of exploiting a stack-based buffer overflow.

By combining the random canary with the initial stored rip address in a XOR operation, the XOR canaries try to thwart these kinds of attacks, except when the program presents a memory leak vulnerability, both the XORed value and the content of RIP can be exposed to the attacker. In real world, this mitigation is implemented using a combination of the XOR Canary, the Random Canary and the Terminator Canary, but another kind of disadvantages reveals. Consider a canary formed by a random value and a terminator character, it is clear it will have a reduced entropy compared to a completely random canary as the size of random bytes reduces also which leads to an easy to guess Canary value.

To visualize the role of canaries (XORed, Terminator and Random) in the protection of the saved return address, Figure 35 shows the memory layout of a program using one of this mechanism before the stack get corrupted, and Figure 36 highlights how the corruption is detected.

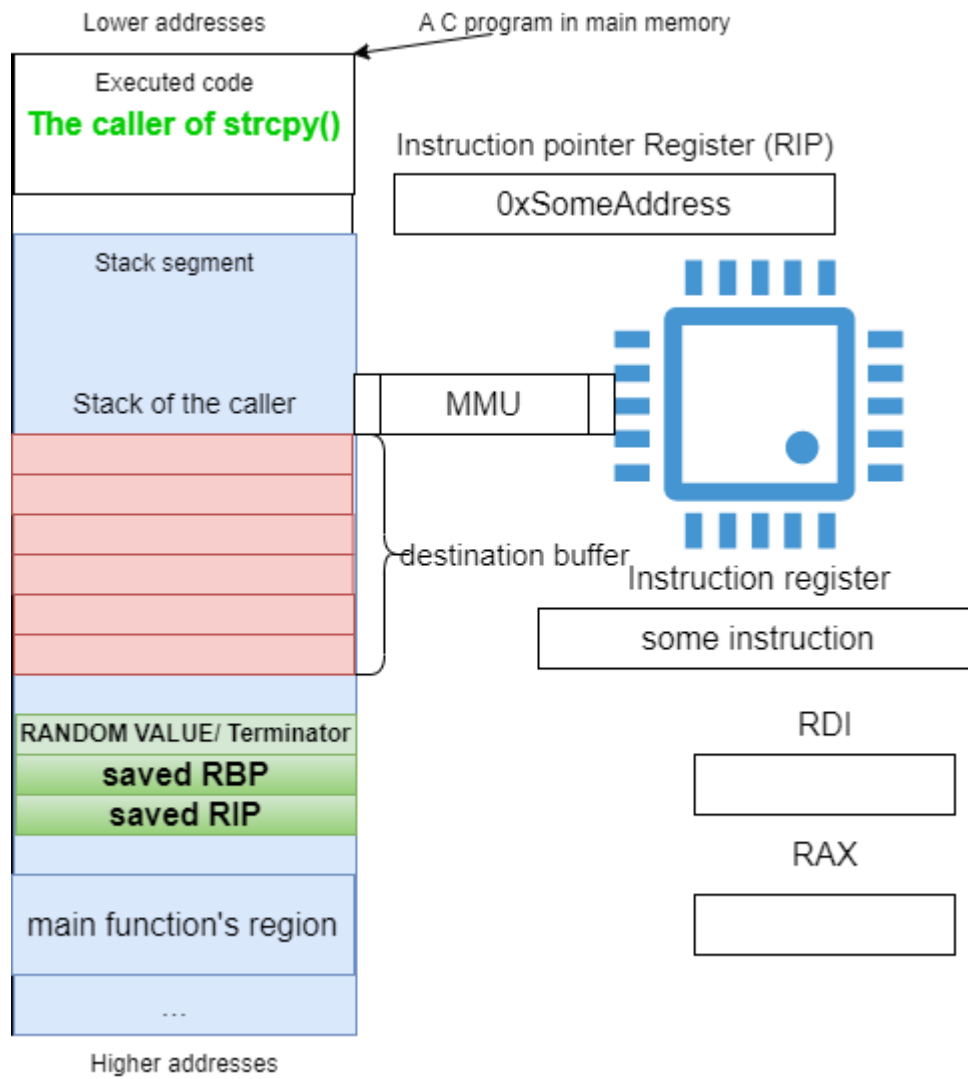


Figure 35: Stack frame before the corruption

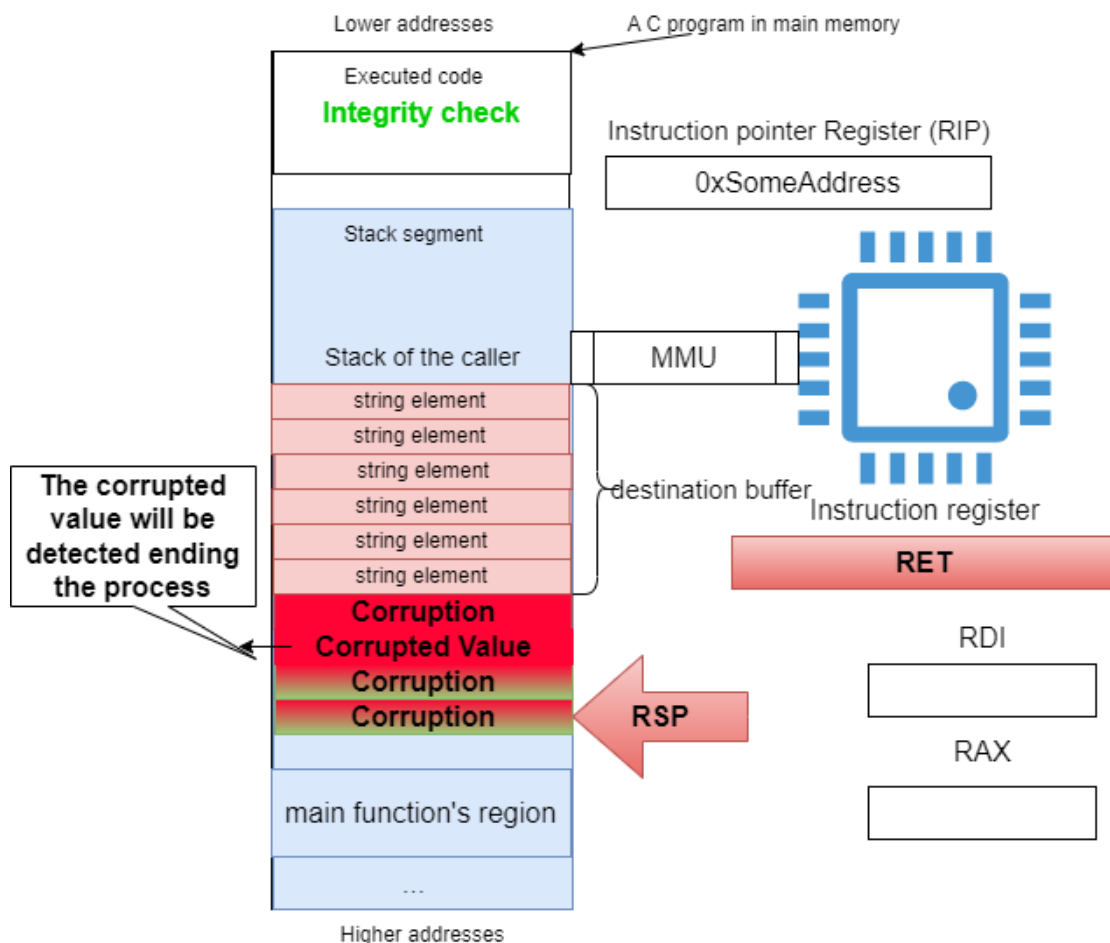


Figure 36: Detected memory corruption

The program invoking `strcpy` didn't handle the case of out-of-bounds memory writing, so the attacker-controlled source buffer was successfully copied into the destination buffer, corrupting the Random (XORed or Terminator) value adjacent to the destination buffer of the `strcpy` function. The Canary mitigation, as the subsequent sections show, includes in the mitigated program a way to check for the integrity of the canary value before returning to the caller. In this case, the function invoking `strcpy` will detect the corruption of the Canary upon returning to its caller, and the program immediately stops.

2.3.5.2 Implementation of stack canaries in GNU/Linux systems

Linux systems that use the GNU C library and `gcc` (i.e., most of the Linux systems) implement stack canaries as a collaboration between the kernel, the compiler and the C library. The workflow is as follows [17]:

1. During each `execve()`, the kernel places a random value in the stack of the new allocated virtual memory;

2. The C runtime initialization functions that come with the GNU libc use this value to compute the canary and place it in a well-known location in the process's memory;

3. the function prologue generated by gcc takes this global canary and pushes it on the stack; the function epilogue checks if the local canary matches the global one, and aborts the process if they differ. We can already make some considerations.

The `execve` is a system call used to load a binary and execute the program it holds; it comes always after the `fork` system call to replace the child process with a new program. As the Canary value changes only after an `execve`, it is always the same for the entire lifetime of a process.

There is a new canary only when `execve()` is called: a `fork()`ed process will use the same canary as its parent.

This implementation also presents some limitations, basically, the Canary exists in many places in memory. A global copy of the Canary is present in a data-structure dedicated for the whole program after an `execve` of that program, and at any moment a function call is introduced, the called function will copy the Canary from the global data-structure to its stack frame. Due to the nature of working of the stack as discovered in chapter 1, no data is really deleted when a stack frame is released, thus, an attacker can read the value of the Canary from parts on the stack different from the frame of the flawed function (i.e., the function using a routine lacking of bounds checking). However, the mitigation still provides a degree of protection, as the attacker has to exploit more than one bug in order to bypass it.

2.3.5.3 *The kernel*

When the user puts the pathname of a program in the shell, the shell will execute an `execve` system call specifying the invoked program, and its environment variables. Upon the loading of the program, the Kernel will put a 16 bytes Random value (generated using a pseudorandom number generator) just above the environment string [17], and uses an auxiliary vector to indicate the address of those Random bytes [17.a]

Auxiliary vectors are vector structures that take up two stack lines each, they provide kernel level information to user-processes. The first line contains a numeric "tag" that identifies the type of information contained into the second line; the `AT_RANDOM` tag (value 25, hex 19) is the one we are interested in; the second line of the entry with this tag contains the pointer to the random bytes. The kernel pushes this data structure onto the process stack, immediately below the environment array. This data structure contains various information about the process and the program and is primarily used by the dynamic loader. The following is an example of such auxiliary vector [17.b]:

```
AT_SYSINFO_EHDR: 0x7fff35d0d000
```

```
AT_HWCAP:      bfebfbff
```

AT_PAGESZ:	4096
AT_CLKTCK:	100
AT_PHDR:	0x400040
AT_PHENT:	56
AT_PHNUM:	9
AT_BASE:	0x0
AT_FLAGS:	0x0
AT_ENTRY:	0x40164c
AT_UID:	1000
AT_EUID:	1000
AT_GID:	1000
AT_EGID:	1000
AT_SECURE:	0
AT_RANDOM:	0x7fff35c2a209
AT_EXECFN:	/usr/bin/sleep
AT_PLATFORM:	x86_64

2.3.5.4 The GNU C library

The GNU C library [14] contains some object files that are linked with all programs by default. The shared objects implement many frequently used routines such as initialization routines which execute at the startup-time of a process, and clean-up routines that execute at the end of any process before it exits. The `_start` One of the startup routines is called `_start`, it is the first executed routine in the lifetime of a process, it implements a small assembly program that calls `_libc_start_main()`, function implemented in C language in the standard C library. The role of this function is to perform some initializations and passes control to the original program's main function. When a program is compiled to have the stack canary mitigation, those initialization routines shall copy the least significant bytes from the kernel-provided random bytes to form the canary value. It depends on the underlying platform; 4 bytes will be copied in case of 32bit architecture and 8 bytes otherwise [17]. The copied parts shall be merged with a terminator character, by replacing the least significant byte by a null character, to end up by a Canary that combines Random bytes and a Terminator byte. This will lead to an easy to guess Canary in the case of 32bit architecture, as the Canary contains only 3 Random bytes.

Upon the program loading, the kernel puts the 16 random bytes at a Thread-Local Storage, a per thread data-structure called Thread Control Block. This data-structure can be corrupted if there in the occurrence of a buffer overflow attack

when the buffer is adjacent to the TCB location, adding a new risk to the canary [17]. Basically, initialization routines put the address of this data structure into fs register (segment selector register) upon the constitution of the Canary value. So, whenever the global Canary value is needed its address will be calculated using this register.

2.3.5.5 The gcc compiler

The gcc compiler will add canary support to the compiled program if the stack-protector option is enabled. In current Linux distributions, this is enabled by default and can be disabled by adding the `-fno-stack-protector` option to the gcc command line.

The following options come from GCC4.9.3 Manual:

<code>-fstack-protector</code>

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

<code>-fstack-protector-all</code>

Like `-fstack-protector` except that all functions are protected.

<code>-fstack-protector-strong</code>

Like `-fstack-protector` but includes additional functions to be protected — those that have local array definitions, or have references to local frame addresses.

When canaries are enabled, the prologue of canary-protected functions becomes:

```
1  ; save the old frame pointer
2  push rbp
3  ; (maybe push other registers)
4  ; create the new frame pointer
5  mov rbp, rsp
6  ; reserve space for the local variables + the canary
7  sub rsp, x
8  ; copy the global canary in the current frame
9  mov rax, QWORD PTR fs:0x28
10 mov QWORD PTR [rbp-8], rax
```

Lines 1–7 contain a standard prologue, except for the need to reserve space for the canary in addition to the local variables. Lines 9 and 10 are new: line 9 reads the global canary from offset 0x28 in the TCB and line 10 copies the canary just above the saved frame pointer. Note that, if the compiler has to save other registers

besides the old frame pointer (see the comment at line 3), the canary will be stored above them. The canary protected epilogue is:

```

1  ; compare the global canary with the local copy
2  mov rax, QWORD PTR [rbp-8]
3  sub rax, QWORD PTR fs:0x28
4  je good
5  ; abort if the local canary has been modified
6  call __stack_chk_fail@plt ; doesn't return
7 good:
8  ; restore old rsp and rbp
9  leave
10 ; return to the caller
11 ret

```

Lines 1–7 are new, while the others are nothing more than the standard epilogue. The new instructions add a bit of overhead to the function, so gcc only adds them where it thinks they are really needed. Basically, only in functions that declare sufficiently large array variables. The `__stack_chk_fail` function prints an error message on standard error and aborts the process. For the time being, ignore the strange `@plt` suffix in the function name: It is a reference to the linker-generated Procedure Linkage Table (PLT).

2.4 Alternative Functions: an Example of `strncpy()`

There is an equivalent to the function `strcpy()` which do consider the size of the destination buffer when copying strings in the stack.

The `strncpy()` function is similar to `strcpy()` function, except that at most `n` bytes of the source (`src`) buffer are copied. If there is no NULL character among the first `n` character of the source buffer, the string placed in the destination buffer (`dest`) will not be NULL-terminated. If the length of `src` is less than `n`, `strncpy()` writes an additional NULL characters to `dest` to ensure that a total of `n` characters are written. Syntax:

```
char *strncpy( char *dest, const char *src, size_t n )
```

`src`: The string which will be copied.

`dest`: Pointer to the destination array where the content is to be copied.

`n`: The first `n` character copied from `src` to `dest`.

The next part describes an implementation of a solution to statically replace the `strcpy()` function with `strncpy()` in the executable file without needing the source code, even if the executable is stripped (i.e., a symbol-free binary).

Patching a stripped binary will require that the vulnerable function is dynamically linked to the program, this is because symbols of statically linked objects can't persist after running a strip command against the binary.

2.4.1 A proposed solution: Substituting the Flawed Function

2.4.1.1 *Problem and solution*

Though it has some limitations, stack canary mitigation is considered as a robust defense against stack's buffer overflow bugs. This mitigation requires that the source code of the vulnerable program is in our hand, which is not always the case. The bug can be detected in proprietary applications, old libraries or out-of-the-shelf software when no source code is available.

We proposed a solution based on a binary rewriting approach. Static binary rewriting has many important applications in software security and systems such as hardening, repair, patching, instrumentation, and debugging. While many different static binary rewriting tools have been proposed, most rely on recovering control flow information from the input binary. Control flow refers to the order in which the instructions of a program are executed. It determines how a program moves from one statement to another, based on specific conditions and decisions.

The recovery step is necessary since the rewriting process may move instructions, meaning that the set of jump targets in the rewritten binary (e.g., the content of memory locations pointed to by the operands of control-flow instructions) needs to be adjusted accordingly.

Since the static recovery of control flow information is a hard problem because of the necessity of the manual efforts it needs, most tools rely on a set of simplifying heuristics or assumptions, such as specific compilers, specific source languages, or binary file meta information.

However, the reliance on assumptions or heuristics tends to scale poorly in practice, and most state-of-the-art static binary rewriting tools cannot handle very large/complex programs such as web browsers.

In this work we use E9Patch, a tool that can statically rewrite x86_64 binaries without any knowledge of control flow information [19]. e9patch is control-flow agnostic and it doesn't depend on any heuristics. We will use this tool as a framework to implement our solution to stack buffer overflow vulnerability, caused by the use of functions that lack bounds checking; And applying the solution to replace strcpy() function with strncpy() function in stripped binaries to give a concrete use-case. To do so, E9Patch develops a suite of binary rewriting methodologies—such as instruction punning, padding, and eviction—that can insert jumps to trampolines without the need to move other instructions [19]. Since our approach preserves the set of jump targets, the need for control flow recovery and related heuristics is eliminated. As such, E9Patch is robust by design, and can scale to very large (>100MB) stripped binaries including the Google Chrome and FireFox web browsers. E9Patch operates at a low level, directly manipulating instructions within a binary file. Its patching process involves several steps. The process can be automated,

either by using its default front-end (e9tool) or by developing a custom frontend. First, e9patch takes an unpatched binary as input, along with disassembly information detailing instruction locations and sizes, as well as the specific patch locations. Additionally, it utilizes trampoline templates, which are code snippets containing the desired functionality to be inserted.

The patching process begins with the selection of patching tactics, where E9Patch attempts a sequence of strategies (Baseline B1, B2, T1, T2, T3) for each patch location. These tactics prioritize efficiency and coverage. Baseline tactics (B1 and B2) involve directly replacing instructions with jumps to trampolines (B1 tactic), or using punned jumps to conserve space (B2 tactic). If these tactics fail, E9Patch employs more advanced techniques like padded jumps, successor eviction, or neighbor eviction to ensure successful patching.

For the x86_64, B1 tactic is implemented using the relative near jump (`jmpq rel32`) instruction. Here `rel32` is a 32bit signed integer that is added to the program counter (`%rip`) in order to orient the jump. The relative near jump instruction is five bytes long, including one byte for the opcode (`0xe9`) and four bytes for the `rel32` value. A patch location instruction that is greater-than-or-equal-to five bytes can be directly replaced, but complications arise when the patch location instruction is smaller than five bytes. To deal with patch location's size limit B2, T1, T2 and T3 tactics are conceived. Those extra tactics are out of the scope of our work, since our patch locations are call instructions to the `strcpy()` or any other flawed function.

ELF rewriting is a key aspect of E9Patch's functionality. It patches instructions in place, replacing targeted instructions with jumps to corresponding trampolines, and appends new data such as trampoline and instrumentation code to the end of the binary. Additionally, it integrates a loader at the entry point to map trampoline pages into the virtual address space during program loading.

The output of E9Patch is a rewritten binary, where the desired patches have been applied. This modified binary serves as a drop-in replacement for the original, requiring no additional dependencies or configuration. Essentially, E9Patch skillfully manipulates instructions and memory layout to achieve efficient and scalable binary rewriting without the complexities of control flow analysis.

Since our patch will target call instructions to the `strcpy()`, our patch location takes 5 bytes in memory (i.e., the size of a call instruction) which is large enough to hold a jump instruction, so replacing the call to `strcpy` with a jump instruction that address a safe version of `strcpy()` (e.g., `strncpy()`) is possible using B1 tactic.

B1 tactic replaces each patch location instruction with a jump instruction that redirects control flow to a trampoline that implements the patch. In our case, a trampoline means a snippet of code that:

1. Save any necessary registers
2. Prepare the arguments for the safer version of the flawed function
3. Call the function that replaces the flawed function

Consider the following instruction which is a call to the dynamically linked `strcpy()` function:

BINARY REPRESENTATION	ASSEMBLY REPRESENTATION
e8 db fe ff ff	call 401030 <strcpy@plt>

To substitute `strcpy()` function with its safer version statically in the binary of the vulnerable application, we need simply to substitute this instruction with a jump instruction to the trampoline which calls `strncpy()`. This requires the existence of the code that implements the new function (e.g., the patch) at runtime, either statically linked or dynamically linked with the patched binary (e.g., the output of the tool). So, when inspecting the same location in the patched binary we would find something like this:

BINARY REPRESENTATION	ASSEMBLY REPRESENTATION
e9 ab 3e 00 00	jmp 405000 <__TMC_END__+0xfd0>

A jump instruction to a trampoline that is already hardcoded by the tool in the binary.

2.4.1.2 E9 tool

E9Tool is the default frontend for E9Patch [21]. E9Tool translates high-level patching commands (i.e., what instructions to patch, and how to patch them) to low-level commands for E9Patch. The basic usage of E9Tool is as follows:

`$ e9tool -M MATCH -P PATCH binary`, Where:

- *binary* is the binary to patch (executable or shared object)
- -M MATCH specifies which instructions in *binary* to patch
- -P PATCH specifies how matching instructions should be patched

After rewriting, the patched binary will be written to `a.out` (for executables) or `a.so` (for shared objects) by default. For example, the following command will instrument all jump instructions in the `xterm` binary. Whenever the jump instruction is executed a message, indicating the execution of jump, is printed at the console:

`$ e9tool -M jmp -P print xterm`

E9tool frontend communicate with the backend, `e9patch` through a client server architecture. The E9Patch tool uses the JSON-RPC (version 2.0) as its API. Basically, the E9Patch tool expects a stream of JSON-RPC messages which describe which binary to rewrite and how to rewrite it. These JSON-RPC messages are fed from a frontend tool, such as E9Tool, but this design means that multiple different frontends can be supported. The choice of JSON-RPC as the API also means that the

frontend can be implemented in any programming language, including C++, python or Rust.

By design, E9Patch tool will do very little parsing or analysis of the input binary file. Instead, the analysis/parsing is left to the frontend, and E9Patch relies on the frontend to supply all necessary information in order to rewrite the binary. Specifically, the frontend must specify:

- The file offsets, virtual addresses and size of instructions in the input binary.
- The file offsets of the patch location.
- The templates for the trampolines to be used by the rewritten binary.
- Any additional data/code to be inserted into the rewritten binary.

The main JSON-RPC messages are:

- Binary Message: begins the patching process. It must be the first message sent to E9Patch. The message specifies the type of the file (.so or executable) and the path name of the file.
- Trampoline Message: Used to specify the template of the trampoline when a patch location is matched.
- Reserve Message: Used to reserve memory sections for code and data in the output file.
- Instruction Message: The message specifies a single instruction in the input file, defines the virtual address of the instruction, its size, and its offset in the binary.
- Patch Message: instruct e9patch to patch an instruction already declared by the instruction message, it also specifies the trampoline template to use.
- Options Message: passes command-line arguments through a JSON message.
- Emit Message: Ends the process by specifying the name and the type of the output file.

This work will rely on the capabilities provided by the default frontend tool (e9tool) to implement our solution instead of using a custom frontend tool.

2.4.1.3 *Matching the patch location*

To replace a function known to have a stack's buffer overflow bug, we first need to identify its location, specifically the file's offset of the call instruction that changes control flow to the block of that function. In the context of C language programming, the call instruction may refer to either the address of the flawed function if it is statically linked, or the address of a Procedural Linkage Table (PLT) routine if the function is dynamically linked (see section 1.3.2.1).

E9tool provides a matching language which specifies what instructions should be patched by the corresponding patch. Matchings are specified by the (`--match MATCH`) or (`-M MATCH`) command-line option. The form of a matching (*MATCH*) is a Boolean expression of *TESTS* using a specific high-level grammar. The user can combine a set of tests using common logical operators.

Tests will be verified against every instruction in the input binary, if they return a true Boolean value, this instruction is considered as a patch location.

A test can be formed using variables. A variable evaluates to some specific property/attribute of the underlying instruction, defined using the following grammar:

VARIABLE ::= [*SPECIFIER* .] *ATTRIBUTE*

Two important attributes are used in our use of the tool, the first is the *call* attribute, this is a Boolean attribute evaluated as true for call instructions and as false otherwise. The second attribute is *target*, this has an integer type, it can be compared with the address of a symbol statically known in the binary. Instructions that call or jump to the specified target will be matched.

Thus, matching an instruction that call `strcpy()` function require a matching expression with the following form:

'call and target == &strcpy'

The '`&`' symbol used in this matching expression will be parsed by the frontend e9tool as an address operator, *&Name* is specified in the documentation of the tool as the runtime address of the named section/symbol/PLT/GOT entry.

2.4.1.4 The patching language

The patch language specifies how to patch matching instructions from the input binary. Patches are specified using the (`--patch PATCH`) or (`-P PATCH`) command-line option, and must be paired with one or more matchings. The basic form of a patch (*PATCH*) uses the following high-level grammar:

PATCH ::= [*POSITION*] *TRAMPOLINE*

POSITION ::= before

| replace

```

| after
TRAMPOLINE ::= empty
| break
| trap
| exit(CODE)
| signal(SIG)
| print
| CALL
| if CALL break
| if CALL goto
| plugin(NAME).patch()

```

A patch is an optional position followed by a trampoline. The trampoline represents code that will be executed when control-flow reaches the matching instruction. The trampoline can be either a built-in trampoline, a call trampoline, or a trampoline defined by a plugin.

The position specifier can take one of the following values:

- **before:** The trampoline will be executed before the matching instruction. That is, the trampoline is an instrumentation. Which means, the matched instruction will be executed after the execution of all the instructions forming the trampoline.
- **replace:** The trampoline replaces the matching instruction. In this case the instruction that is considered as a patch location will be omitted in the patched binary.
- **after:** The trampoline is executed after the matching instruction.

2.4.1.5 *Built-in trampolines*

Those are the trampolines provided by e9tool and used for multiple purposes, following is a list of built-in trampolines and their purposes:

- **empty:** is the empty trampoline with no instructions. Control-flow is still redirected to/from empty trampolines, and this can be used to establish a baseline for benchmarking.
- **break:** immediately returns from the trampoline back to the main program.
- **trap:** executes a single TRAP (int3) instruction.
- **exit(CODE):** will immediately exit from the program with status CODE.

- `signal(SIG)`: will raise signal SIG in the current thread (equivalent to `kill(gettid(), SIG)`).
- `print`: will print the assembly representation of the matching instruction to `stderr`. This can be used for testing and debugging.

2.4.1.6 Custom trampolines

By developing a custom frontend tool that uses the API of `e9patch`, one can have a absolute flexibility on the way of developing trampolines. This will also make the tool able to be integrated in development environment, testing projects and so on.

The `e9tool` frontend provides multiple options to develop and integrate trampolines, thus, to implement a patch. One possibility is to use something called plugins, an `E9Tool` plugin is a shared object that exports specific functions. These functions will be invoked by `E9Tool` at different stages of the patching process. Some tasks, such as disassembly, will be automatically handled by the `E9Tool` frontend.

The `E9Tool` plugin API is simple and consists of the following functions:

`e9_plugin_init(const Context *cxt)`: Called once before the binary is disassembled.

`e9_plugin_event(const Context *cxt, Event event)`: Called once for each event (see the `Event` enum).

`e9_plugin_match(const Context *cxt)`: Called once for each match location.

`e9_plugin_code(const Context *cxt)`: Called once per trampoline template (code).

`e9_plugin_data(const Context *cxt)`: Called once per trampoline template (data).

`e9_plugin_patch(const Context *cxt)`: Called for each patch location.

`e9_plugin_fini(const Context *cxt)`: Called once after all instructions have been patched.

Each function takes a `cxt` argument of type `Context` defined in `e9plugin.h`.

Plugins are invoked using the `E9Tool` `--match/-M` or `--patch/-P` options. For example:

```
$ g++ -std=c++11 -fPIC -shared -o myPlugin.so myPlugin.cpp -I src/e9tool/
$ ./e9tool -M 'plugin(myPlugin).match()' > 0x333' -P 'plugin(myPlugin).patch()' xterm
```

Where `myPlugin.so` is the shared object which defines the functions that `e9tool` will use in the course of each phase of the rewriting process.

However, our use of the tool takes advantage of a simple mechanism called call trampoline. A call trampoline calls a user-defined function that can be implemented

in a high-level programming language such as C or C++. Call trampolines are the main way of implementing custom patches using E9Tool. The syntax for a call trampoline is as follows:

`CALL ::= FUNCTION [ABI] ARGS @ BINARY`

`ABI ::= < clean | naked >`

`ARGS ::= (ARG , ...)`

The call trampoline specifies that the trampoline should call function `FUNCTION` from the binary `BINARY` with the arguments `ARGS`. To use a call trampoline, the tool's manual dictates:

1. Implement the desired patch as a function using the C or C++ programming language.
2. Compile the patch program using the special `e9compile.sh` script to generate a patch binary.
3. Use the E9Tool frontend to call the patch function from the patch binary at the desired locations.

E9Tool will handle all of the low-level details, such as loading the patch binary into memory, passing the arguments to the function, and saving/restoring the CPU state.

The `e9compile.sh` script is a `gcc` wrapper that ensures the generated binary is compatible with E9Tool.

Call trampolines support two Application Binary Interfaces (ABIs).

- `clean`: saves/restores the CPU state and is compatible with C/C++, this ABI causes the same behavior defined by system V ABI (see section 1.4.2).
- `naked`: saves/restores registers corresponding to arguments only and give more flexibility to the trampoline developer.

2.4.1.7 *Call Trampoline*

With Call Trampoline method, a trampoline is a function coded in the C or C++ language. This function may or may not be fed data. The data can be static, fixed values determined when the patch is programmed, or it can depend on the execution state of the patched program.

`e9tool` allows passing arguments by value to functions called by call trampolines. This is achieved by specifying the argument types directly. For instance, the syntax to pass the current value of the instruction pointer (`%rip`) to a function looks like this:

```
$./e9tool -M ... -P 'func(rip)@example' xterm
```

In this example, `func` is called with the value of `%rip` as an argument, and the corresponding C function can be defined as follows:

```
void func(const void *rip) {
    // RIP register value as an argument
}
```

The call trampoline can support up to eight arguments, and these arguments can be of various types such as integers, strings, addresses, and more.

e9tool also supports passing arguments by pointer, allowing for more dynamic interaction with the program's state. This feature is particularly useful for modifying values at runtime. For example, consider a function that increments the value pointed to by a pointer:

```
void inc(int64_t *ptr) {
    *ptr += 1;
}
```

We can define a call trampoline to invoke this function and pass the address of a register (e.g., %rax) as follows:

```
$ e9compile.sh example.c
$ e9tool -M ... -P 'inc(&rax)@example' xterm
```

In this case, the value of %rax will be incremented each time inc is called. Thus, every time an instruction is matched and the rewriting phase is reached the content of %rax at that moment will be incremented. The behavior of pointers depends on the operand type:

- Immediate operands:

point to constant values stored in read-only memory. For example, a constant value directly embedded in the matched instruction. The address of the operand 5 in the instruction: mov eax, 5 can be referenced by pointer when that instruction is matched. This can be done by passing the keyword: &mem[2] as an argument to the trampoline.

- Register operands:

Point to temporary locations holding the register values. This is a reference to a CPU register that holds data. For example, in the instruction mov eax, ebx, both eax and ebx are register operands. The only way to manipulate the value of a register at runtime is to pass the address of that register as an argument to the trampoline.

- Memory operands:

Provide the exact runtime pointer value calculated by the operand itself. This is a reference to a location in memory. The operand specifies an address from which

data is fetched or to which data is stored. For example, in the instruction `mov eax, [ebx]`, `[ebx]` is a memory operand referring to the memory address contained in `ebx`.

2.4.1.8 Implementation

It is common to find that the ELF executable file of a C program is stripped. ELF stripping is the process of removing unnecessary information from an Executable and Linkable Format (ELF) file, such as debugging symbols and relocation information, to reduce its size and improve performance. This is typically done using the *strip* Linux command.

If `strcpy()` function (or a similar function) is dynamically linked to the vulnerable program, then our solution would work even if the executable ELF file is stripped (e.g., the function symbol is presented in the PLT section). If the function is statically linked to the program to-be-patched, then the ELF executable file must not be stripped in order to patch it.

Using `e9patch` we can substitute whatever unsafe function lacking bounds checking. For the sake of a concrete example, we chose to apply the method on a specific function. If applied automatically, this method does not protect against the corruption of local variables. Applying `e9patch` automatically means that no manual effort is needed, the tool will replace `strcpy()` with `strncpy()`, providing to `strncpy` a size which is also computed automatically, this is the value contained in `rbp` minus the value contained in `rdi` (see Figure 38).

Though the implementation only reinforces the calling conventions, it can maintain the integrity of local variables if we accompany it with static analysis of the program. For example by extracting the size of each destination buffer used by `strcpy()` function one can implements a trampoline that replaces the function with `strncpy()` function and specifies the appropriate size as an argument to `strncpy`.

This is an efficient way to enforce the integrity of C program's control flow.

Consider the program in Figure 37 which uses the unsafe `strcpy()` function:

```
#include <stdio.h>

#include <string.h>

void printName(char* buffer) {
    char name[16];
    //la fonction a remplacer
    strcpy(name, buffer);
}
```



```
printf("Hello %s\n", name);  
}  
  
int main(int argc, char* argv[]) {  
if(argc > 1) printName(argv[1]);  
return 0;  
}
```

Figure 37: The vulnerable program

The tool will replace `strcpy(dest, src)` function with the safer `strncpy(dest, src, size)` function, which is considered more secure due to its explicit size determination.

The patch is created in the form of a function (see Figure 38) and compiled in a specific way compatible with the e9tool. This use of the tool is explained in section 2.4.1.8, a method known as Call Trampoline.

```
#include "/e9patch/examples/stdlib.c"  
  
void patch (void *rbp, void *rdi, char  
*rsi){  
long size= (long) rbp - (long) rdi;  
strncpy(rdi,rsi,size-1);  
}
```

Figure 38: Trampoline implementation

The values of the RBP and RDI registers are passed by value to calculate the available space between the buffer and the address where the previous frame address is stored.

The tool comes with a compiler wrapper, a script that directs GCC to generate a compatible ELF file.

The compilation is done by the following command in a Linux environment:

```
$ e9compile.sh patch.c
```

After preparing the patch, we instructed e9tool to match a call to `strcpy()` and replace it with the trampoline following the `-P` flag. This is done by the following command:

```

${e9}/e9tool -s -o file -O0 -M 'call and target == &strcpy' -P 'replace
patch(rbp,rdi,rsi)@newpatch' --debug ../strippednoCanari

```

Where `${e9}/e9tool` substituted by the file path of the tool.

-s flag instruct e9tool to hardcode the trampoline in the output file, instead of using a custom loader, so, when disassembling the final result we can see the difference with the former version of the ELF file.

-O0 option, disables any optimization technique to reduce the size of the file.

The result is a program (i.e., the ELF named file) identical to the original program except that the function `strcpy()` is replaced by its equivalent `strncpy()`.

To illustrate the difference between the input file and the output file, Figure 39 highlights the patch location in the original file, that will be matched by e9tool.

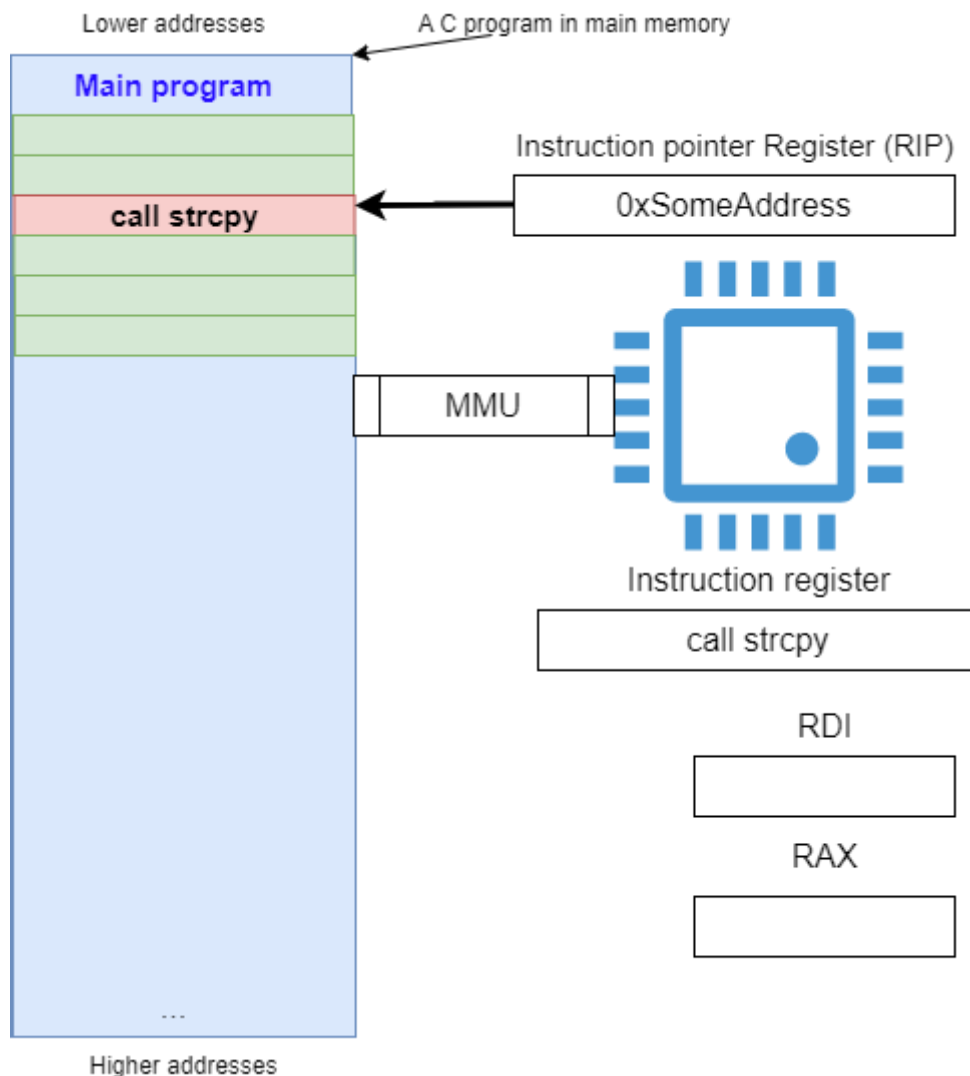


Figure 39: The main program before applying the patch process

After being matched, the tool will rewrite the call instruction with a jump to the trampoline which is appended at the end of the binary along with the implementation of the function `strncpy`.

The tool also changes the entry point of the program to a specific routine, responsible for allocating space in memory for the trampoline and the `strncpy` code [19].

In the output binary, a jump instruction replaces the call instruction, so when RIP will point to this memory location, control flow will be passed to the trampoline. The trampoline starts by saving the state of the CPU (register's values) unless a *naked* ABI is specified as argument to the trampoline. The trampoline sets the arguments for the `strncpy` function, including the size to be copied which is the distance between the location pointed to by RDI and the location pointed to by the RBP register, calculated at runtime. Then, a call instruction in the trampoline block changes control flow to the block of `strncpy` function. After the execution of `strncpy` code, control flow returns to the trampoline to restore the value of any saved register and returns to the main program (see Figure 40).

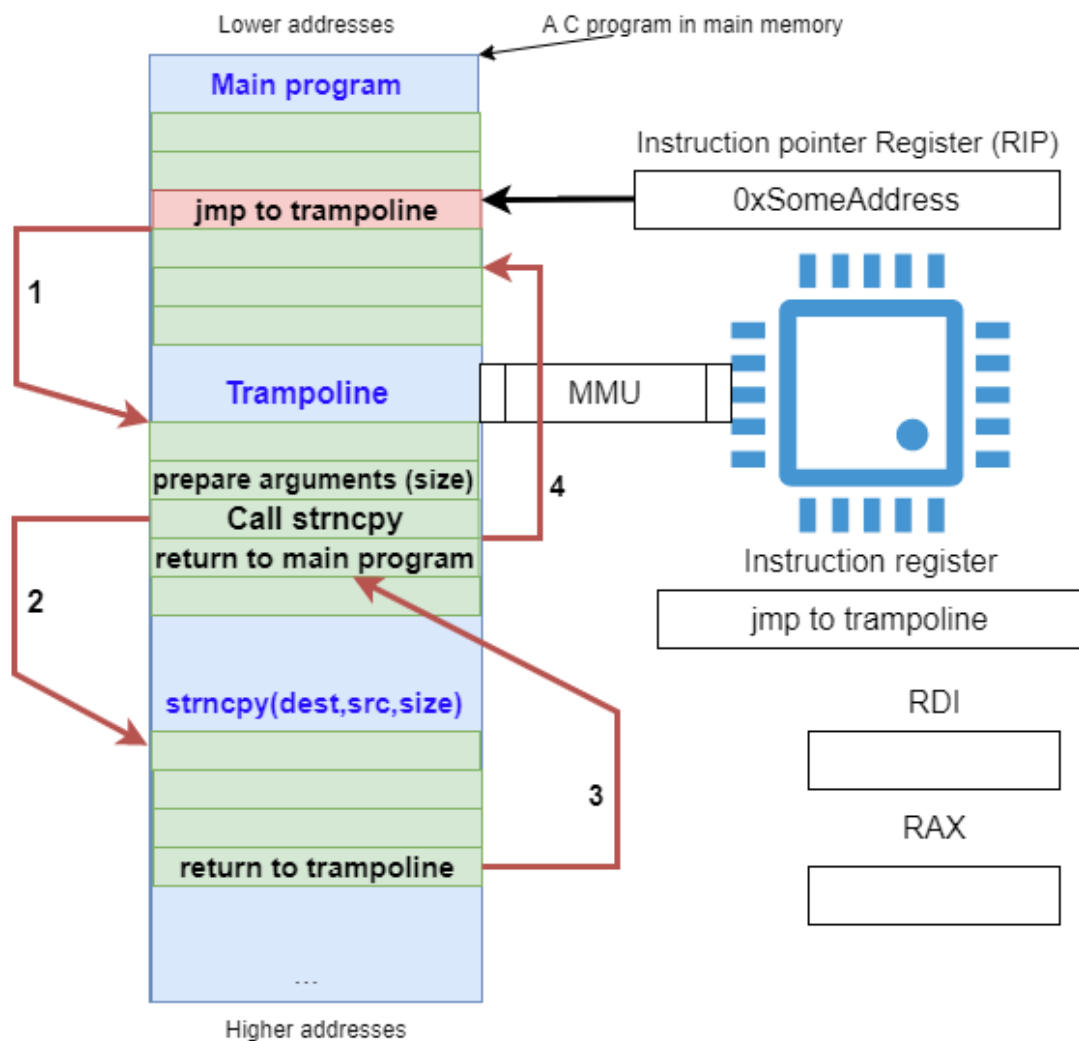


Figure 40: The workings of the patched program

Figure 41 shows the instruction in the block of printName function that calls the function strcpy.

```

0000000000401136 <printName>:
401136:    55                push    rbp
401137:    48 89 e5          mov     rbp, rsp
40113a:    48 83 ec 20        sub     rsp, 0x20
40113e:    48 89 7d e8        mov     QWORD PTR [rbp-0x18], rdi
401142:    48 8b 55 e8        mov     rdx, QWORD PTR [rbp-0x18]
401146:    48 8d 45 f0        lea     rax, [rbp-0x10]
40114a:    48 89 d6          mov     rsi, rdx
40114d:    48 89 c7          mov     rdi, rax
401150:    e8 db fe ff ff    call    401030 <strcpy@plt>
401155:    48 8d 45 f0        lea     rax, [rbp-0x10]
401159:    48 89 c6          mov     rsi, rax
40115c:    bf 10 20 40 00    mov     edi, 0x402010
401161:    b8 00 00 00 00    mov     eax, 0x0
401166:    e8 d5 fe ff ff    call    401040 <printf@plt>
40116b:    90                nop
40116c:    c9                leave
40116d:    c3                ret

```

Figure 41: The assembly representation of printName before the patching process

The output of e9patch rewrites this instruction by a jmp instruction to the trampoline as Figure 42 shows:

```

0000000000401136 <printName>:
401136:    55                push    rbp
401137:    48 89 e5          mov     rbp, rsp
40113a:    48 83 ec 20        sub     rsp, 0x20
40113e:    48 89 7d e8        mov     QWORD PTR [rbp-0x18], rdi
401142:    48 8b 55 e8        mov     rdx, QWORD PTR [rbp-0x18]
401146:    48 8d 45 f0        lea     rax, [rbp-0x10]
40114a:    48 89 d6          mov     rsi, rdx
40114d:    48 89 c7          mov     rdi, rax
401150:    e9 ab 3e 00 00    jmp     405000 <__TMC_END__+0xfd0>
401155:    48 8d 45 f0        lea     rax, [rbp-0x10]
401159:    48 89 c6          mov     rsi, rax
40115c:    bf 10 20 40 00    mov     edi, 0x402010
401161:    b8 00 00 00 00    mov     eax, 0x0
401166:    e8 d5 fe ff ff    call    401040 <printf@plt>
40116b:    90                nop
40116c:    c9                leave
40116d:    c3                ret

```

Figure 42: A substituted call instruction

Triggering an out-of-bounds memory writing by providing a long input to the original program gave a segmentation fault error mentioned at Figure 43:

```

Test-->../noCanari aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaa
Hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
Test-->

```

Figure 43: Executing the flawed program

Providing the same input to the patched program shows the expected safe behavior:

```

Test-->./a.out aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Hello aaaaaaaaaaaaaaa
Test-->

```

Figure 44: Executing the patched program

2.5 Guidelines for secure programming

MSC24-C is a specific guideline within the CERT C Secure Coding Standard, which provides rules and recommendations for writing secure C code [22]. The CERT C Secure Coding Standard is a set of guidelines developed by the CERT Coordination Center at Carnegie Mellon University to help programmers write more secure and reliable C code. MSC24-C specifically addresses "Be careful with the use of `setjmp()` and `longjmp()`", emphasizing the potential security risks associated with these functions and providing recommendations for using them safely.

This guideline is published online, below are their important advices to programmers in the context of our work:

Do not use deprecated or obsolescent functions when more secure equivalent functions are available. Deprecated functions are defined by the C Standard. Obsolescent functions are defined by this recommendation.

Old	New
Obsolescent Function	Recommended Alternative
<code>bsearch()</code>	<code>bsearch_s()</code>
<code>fprintf()</code>	<code>fprintf_s()</code>
<code>fscanf()</code>	<code>fscanf_s()</code>

Old	New
Obsolescent Function	Recommended Alternative
fwprintf()	fwprintf_s()
fwscanf()	fwscanf_s()
getenv()	getenv_s()
gmtime()	gmtime_s()
localtime()	localtime_s()
mbsrtowcs()	mbsrtowcs_s()
mbstowcs()	mbstowcs_s()
memcpy()	memcpy_s()
memmove()	memmove_s()
printf()	printf_s()
qsort()	qsort_s()
scanf()	scanf_s()
snprintf()	snprintf_s()
sprintf()	sprintf_s()
sscanf()	sscanf_s()
strcat()	strcat_s()
strcpy()	strcpy_s()

Old	New
Obsolescent Function	Recommended Alternative
strerror()	strerror_s()
strlen()	strnlen_s()
strncat()	strncat_s()
strncpy()	strncpy_s()
strtok()	strtok_s()
swprintf()	swprintf_s()
swscanf()	swscanf_s()
vfprintf()	vfprintf_s()
vfwscanf()	vfwscanf_s()
vfwprintf()	vfwprintf_s()
vfwscanf()	vfwscanf_s()
vprintf()	vprintf_s()
vscanf()	vscanf_s()
vsnprintf()	vsnprintf_s()
vsprintf()	vsprintf_s()
vsscanf()	vsscanf_s()
vswprintf()	vswprintf_s()

Old	New
Obsolescent Function	Recommended Alternative
vswscanf()	vswscanf_s()
vwprintf()	vwprintf_s()
vwscanf()	vwscanf_s()
wcrtomb()	wcrtomb_s()
wcscat()	wcscat_s()
wcscpy()	wcscpy_s()
wcslen()	wcsnlen_s()
wcsncat()	wcsncat_s()
wcsncpy()	wcsncpy_s()
wcsrtombs()	wcsrtombs_s()
wcstok()	wcstok_s()
wcstombs()	wcstombs_s()
wctomb()	wctomb_s()
wmemcpy()	wmemcpy_s()
wmemmove()	wmemmove_s()
wprintf()	wprintf_s()
wscanf()	wscanf_s()

Table 45: Deprecated functions and their alternative

2.6 Conclusion

Using functions lacking bounds checking carefully or replace them with secure functions will protect the safety of the stack memory region at an early stage. Thus, cultivating developers of security related aspects is important for developing a secure software.

General conclusion and perspectives

The security of the stack memory must not depend on one line of defense, as we discovered throughout the second chapter, there is no one security measure that fits every situation. It is the combination of a set of defense mechanisms that makes our stack memory robust against attacks. The more defenses are there the more secure the stack will be.

In this work, I believe that the best solution to stack-based buffer overflow caused by uncaredful use of functions lacking bounds checking is to replace the function with one that do consider buffer's bounds. This can be done either manually when the source code is available or using a binary rewriting technique like the one presented at the end of the second section.

References

[d1] Intel® Itanium™ Processor specific Application Binary Interface (ABI) – May 2001

[1] Limited direct execution mechanism: <https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf>

[2] L. Hatton, "Software failures-follies and fallacies," in IEE Review, vol. 43, no. 2, pp. 49-52, 20 March 1997, doi: 10.1049/ir:19970201. keywords: {Software reliability},

[3] Programming with C - <https://www.caluniv.ac.in/academic/LibSc/Study/C-Lang.pdf>

[4] C Programming Language - [https://en.wikipedia.org/wiki/C_\(programming_language\)](https://en.wikipedia.org/wiki/C_(programming_language))

[5] Matthew C. Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In Proceedings of the second international workshop on Computing education research (ICER '06). Association for Computing Machinery, New York, NY, USA, 73–84. <https://doi.org/10.1145/1151588.1151600>

[6] GNU Compiler Collection - <https://gcc.gnu.org/>

[7] Compiler program - <https://en.wikipedia.org/wiki/Compiler>

[8] Linking process, Carnegie Mellon University: Computer Systems: A Programmer's Perspective, chapter 7, <http://csapp.cs.cmu.edu/2e/ch7-preview.pdf>

[9] X86-64 Architecture Guide - <http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html>

[10] objdump(1) - Linux man page - <https://linux.die.net/man/1/objdump>

[11] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V application binary interface, AMD64 architecture processor supplement. May 2009. Draft 0.99.

[12] CS61 2018 Harvard course, Assembly 2: Calling convention - <https://cs61.seas.harvard.edu/site/2018/Asm2/#:~:text=Note%20what's%20missing%3A%20the%20data,only%20accessed%20from%20the%20top.>

[13] Bounds checking - https://en.wikipedia.org/wiki/Bounds_checking

[14] GNU C Library - <https://sourceware.org/glibc/>

[15] GNU Debugger - <https://sourceware.org/gdb/documentation/>

[16] System Memory Management Unit - <https://www.intel.com/content/www/us/en/docs/programmable/683567/21-3/system-memory-management-unit-falconmesa.html>

[17] Stack canaries - G. Lettieri - <https://lettieri.iet.unipi.it/hacking/canaries.pdf>

[17.a] Auxiliary vector in ELF programs, <https://articles.manugarg.com/aboutelfauxiliaryvectors>

[17.b] getauxval() and the auxiliary vector- <https://lwn.net/Articles/519085/?spm=a313e.7916648.0.0.8i0HiO>

- [18] mmap – map pages of memory – <https://pubs.opengroup.org/onlinepubs/009604499/functions/mmap.html>
- [19] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 151–163. <https://doi.org/10.1145/3385412.3385972>
- [20] e9tool – User’s Guide - <https://github.com/GJDuck/e9patch/blob/master/doc/e9tool-user-guide.md>
- [21] e9patch – Programmer’s Guide - <https://github.com/GJDuck/e9patch/blob/master/doc/e9patch-programming-guide.md>
- [22] MSC24-C. Do not use deprecated or obsolescent functions - <https://wiki.sei.cmu.edu/confluence/display/c/MSC24-C.+Do+not+use+deprecated+or+obsolescent+functions>