Automatic Exploitation of Fully Randomized Executables

Austin Gadient
MIT EECS & CSAIL
USA
agadient@csail.mit.edu

Baltazar Ortiz
MIT EECS & CSAIL
USA
baltazar@csail.mit.edu

Ricardo Barrato
Aarno Labs
USA
ricardo@aarno-labs.com

Eli Davis
Aarno Labs
USA
eli@aarno-labs.com

Jeff Perkins
Aarno Labs
USA
jhp@aarno-labs.com

Martin C. Rinard
MIT EECS & CSAIL
USA
rinard@csail.mit.edu

ABSTRACT

We present Marten, a new end to end system for automatically discovering, exploiting, and combining information leakage and buffer overflow vulnerabilities to derandomize and exploit remote, fully randomized processes. Results from two case studies highlight Marten's ability to generate short, robust ROP chain exploits that bypass address space layout randomization and other modern defenses to download and execute injected code selected by an attacker.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Software reverse engineering**;

KEYWORDS

exploit; symbolic execution; taint analysis; information leakage

1 INTRODUCTION

Memory corruption vulnerabilities have been a common target for control flow hijacking attacks for decades [13, 14, 24, 32]. Attackers inject code into a running process, or reuse code that already exists, to subvert the system. Address Space Layout Randomization (ASLR) has proven to be an effective defense against these attacks [29]. While it is possible to apply ASLR to only some parts of the process, modern systems such as Ubuntu 18.04 apply full ASLR to all code and data in a process. This paper addresses full ASLR.

1.1 Marten

We present Marten, a novel end to end automated exploitation system which generates robust control flow hijacking exploits that bypass full ASLR. A cornerstone of Marten is its ability to automatically find and exploit information leakage vulnerabilities to exfiltrate information about the layout of the address space of the target process. This information enables Marten to then generate payloads that successfully exploit buffer overflow vulnerabilities by overwriting code pointers (such as return pointers) to point to selected code sequences in the randomized process. Our current Marten implementation generates robust ROP chains that enable arbitrary command/code execution even in the presence of full ASLR. These exploits work against fully stripped binaries compiled in different environments and running in different varieties of Linux.

To the best of our knowledge, Marten is 1) the first system to automatically find and exploit buffer overread vulnerabilities to derandomize reusable code (i.e., find the addresses of the code in the randomized address space) and 2) the first system to automatically generate control-flow hijacking exploits that download and execute code of the attacker's choice into processes protected with full ASLR.

1.2 Scope

Marten targets open-source software for which source and benign inputs are available. Starting with the benign inputs, Marten automatically deploys a vulnerability discovery algorithm that works with an instrumented version of the application to identify target sites and craft inputs that expose buffer overread and buffer overflow vulnerabilities.

Marten is designed to exploit processes running on remote servers, accessing these processes via their network interfaces. We emphasize that even though Marten works with an instrumented version of the application to discover the vulnerabilities, it generates exploits that work on stripped binary versions of the source code compiled by others and running on remote servers.

Upon discovering a buffer overread vulnerability, Marten crafts an input that causes the target process to send regions of its address space back to the process that sent the input without crashing the target process. Marten then examines the returned data to derandomize addresses (such as the location of standard libraries like `libc`) within the victim process.

Marten leverages this address space layout information to synthesize short ROP chains that achieve the desired goal of the attacker. Our current implementation, for example, synthesizes ROP chains that download and run a program of the attacker's choice from a remote server. These ROP chains are designed to be short enough to fit into a single TCP packet and contain padding that makes them robust against environment differences.

To execute the ROP chain, Marten automatically discovers a buffer overflow vulnerability and generates payloads that use this vulnerability to inject the ROP chain into the running process and exploit it. Once given the source code and benign inputs, Marten is fully automatic — it generates the exploit with no human intervention. We emphasize that the exploits it generates are effective against stripped binaries compiled on different systems than our testing environment.

The exploits generated by Marten bypass more defenses than just full ASLR. In fact, the use of information leakage gives us enough ROP gadgets that we can bypass three other widely deployed defenses such as NX, Full RELRO, and Fortify Source. We discuss these defenses in greater detail in Section 2.

1.3 Case Studies

We apply Marten to exploit four vulnerabilities in two processes. The first process is Dnsmasq-2.77, a commonly used DHCP Server. Dnsmasq-2.77 has CVE-2017-14494, a buffer overread, and CVE-2017-14494, a buffer overflow. Starting with benign inputs, Marten automatically discovers both vulnerabilities. It crafts an input that targets the buffer overread to leak the desired data (an address from `libc`). Working with gadgets from the derandomized `libc` code, Marten crafts a ROP chain that exploits the process. Finally, Marten crafts an input that exploits the buffer overflow to inject the ROP chain into the target process. The final output is a Python file which takes the IPv6 address of the target and automatically executes an exploit that downloads and runs a program from a server that the attacker controls.

The second is Nginx-1.4.0, a popular reverse proxy, compiled using the SSL library OpenSSL-1.0.1. Here Marten automatically discovers CVE-2014-0160 (the infamous Heartbleed buffer overread vulnerability in OpenSSL-1.0.1) and CVE-2013-2028 (a buffer overflow vulnerability in Nginx). As for Dnsmasq, Marten exploits the buffer overread to derandomize `libc`, builds the ROP chain, and exploits the buffer overflow to inject the ROP chain.

For both processes, Marten automatically generates exploits that bypass full ASLR, NX, full RELRO, and Fortify Source. Our exploits are initially generated and tested on Ubuntu. We also test exploits that target stripped binaries compiled on Debian and Red Hat to verify their robustness across different environments.

1.4 Contributions

This paper makes the following contributions:

- **Marten:** It presents Marten, a novel end to end system that automatically targets information leakage and buffer overflow exploits to generate exploits that bypass full ASLR, NX, full RELRO, and Fortify Source. Marten generates these exploits automatically with no human intervention to exploit remote stripped binaries compiled and run in different environments.
- **Information Leakage:** It presents a new algorithm and approach to automate the exploitation of information leakage vulnerabilities, with the information used to derandomize attacker reusable code in remote processes. This approach is built on a mature program instrumentation system that combines taint analysis and targeted symbolic execution to obtain symbolic equations and input byte constraints that enable Marten to generate inputs that exfiltrate derandomization information without crashing the target process.
- **Chain Generation:** It presents an end to end ROP chain generation approach that leverages the availability of derandomized libraries to produce short, effective ROP chains that fit into a single TCP packet and contain enough padding to make the chains effective across different environments.
- **Case Studies:** It presents case studies using Marten to exploit vulnerabilities in three large codebases, Nginx-1.4.0, Dnsmasq-2.77, and OpenSSL-1.0.0. The results highlight Marten's ability to combine buffer overreads and buffer overflows to generate payloads that successfully exploit processes protected with full ASLR (as well as a range of other modern defenses).

1.5 Structure

The remainder of the paper is structured as follows. In Section 2, we provide an overview of Marten and discuss the defenses it circumvents. In Section 3, we provide examples of how our system works when exploiting vulnerabilities in Dnsmasq-2.77 and Nginx-1.4.0 compiled with OpenSSL-1.0.1. Section 4 presents a technical analysis of the instrumentation we place on target programs. Section 5 presents our approach to vulnerability discovery in detail. In Section 6 we present our novel approach to automatically generating information leakage exploits. In Section 7, we present Marten's final exploit generation component and ROP chain generation algorithms. Section 8 discusses related work. Section 9 concludes the paper.

2 OVERVIEW

We discuss challenges incumbent with automatically generating robust exploits that bypass full ASLR, NX and other defenses present on modern systems and present an overview of Marten and how it bypasses these defenses.

2.1 Defenses

This section details the defenses that Marten overcomes. We discuss how they work at a technical level and briefly detail how Marten handles with them. All applications presented in Section 3 were compiled with these defenses.

2.1.1 ASLR. Address Space Layout Randomization (ASLR) [29] randomizes the virtual addresses of a process's memory pages with the goal of eliminating the attacker's ability to locate code in the address space for code re-use attacks like ROP [25]. Originally, ASLR did not apply to the text section of an executable [26]. Today, systems randomize the location of all code and data, including information located in the text section of the executable. Programs with full randomization are said to be Position Independent Executables (PIE) [19] on Linux systems. Marten uses information leakage exploits to bypases this defense.

2.1.2 NX. The non-executable bit (NX) [6] makes it impossible for memory that is initially mapped into a process (such as the stack and heap) to be both executable and writable. Thus, it requires an attacker to reuse code already present in the process to develop an exploit, rather than injecting their own code. Marten handles this defense by using ROP. Note that NX does not prevent an application from mapping in a writable and executable page at runtime. Thus, Marten can use ROP to map in a page as writable and executable and read our shellcode into this page. This mechanism allows for

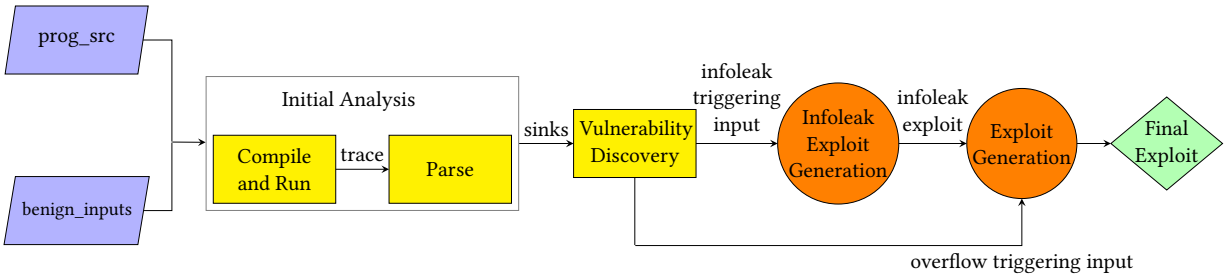

Figure 1: Workflow of Marten. Marten runs initial analysis on two separate seed inputs, one for the information leak and one for the buffer overflow. The execution trace data for each input is sent to vulnerability discovery and Marten mutates the benign inputs until they trigger a bug. Then, Marten finishes the infoleak exploit to spill the desired information from process RAM. This exploit gets sent to finalize a buffer overflow exploit which bypasses full ASLR and other widely deployed defenses.

code execution without being constrained to the gadgets present in a binary or library.

2.1.3 Full RELRO. Relocation Read-Only (RELRO) [18] protects an executable by rearranging where certain data sections are located to prevent overflows into function pointers. Furthermore, full RELRO makes the Global Offset Table (GOT), a location that is full of function pointers to library functions, read only. Thus, an attacker cannot hijack the function pointers located here to take control of code execution. Additionally, an attacker cannot partially overwrite these function pointers to access code located nearby. This is a technique that is commonly used to obtain missing gadgets or bypass partial ASLR. None of our generated ROP payloads attempt to overwrite GOT addresses, allowing us to bypass this protection.

2.1.4 Fortify Source. Fortify Source [20] places extra checks in certain dangerous libraries to ensure they do not result in an overflow. For example, Fortify Source will check at compile time that calls to `memcpy` of a fixed size do not overflow the destination buffer. It is often difficult for the compiler to reason about whether a function is being used unsafely at compile time. This is especially true when the size of a memory transfer is decided by data received at runtime in another function. The exploits generated by Marten in Section 3 all work against binaries compiled with this defense in place.

2.2 Marten

Figure 1 presents the overall workflow of Marten. First, Marten runs initial analysis and vulnerability discovery to find an overread error. It then generates a finely tuned input file which provides Marten with an information leakage exploit. Marten then generates a ROP payload for the target application based on data garnered from the information leakage exploit. Marten reruns the vulnerability discovery algorithm to find a stack buffer overflow. Once this overflow is found, our ROP payload is fit into our vulnerability triggering input using the SMT solver Z3 [5]. This final exploit payload is sent to the target application and used to verify that the exploit works.

2.2.1 Instrumentation. During vulnerability discovery and exploit generation, Marten works with an instrumented version of the application compiled locally. This instrumentation can provide Marten with full symbolic expressions, in terms of program constants and input bytes, for all values computed by the program. To make the collection of these expressions tractable, Marten only computes these expressions for bytes used in *sinks*, i.e., potential target locations within the program. Examples of these target locations include calls to `memcpy` and `recv` where the length or pointers are *tainted*, i.e., influenced by the input. The instrumentation is implemented as an augmented version of LLVM's Dataflow Sanitizer (DFSAN) [9] that supports logging information about operations performed on tainted data as it flows through the program. This log provides a trace of the program's execution after it has received our input. We implement a trace analyzer which operates on these logs to derive the symbolic expressions.

2.2.2 Vulnerability Discovery. Marten uses a goal directed branch enforcement algorithm to find vulnerabilities [27]. Marten iterates through the sinks identified in the initial analysis and try to generate a vulnerability for each one. To find a vulnerability, Marten adds constraints one at a time until it triggers the bug or the solver fails to generate an input file based on the constraints. The main goal of vulnerability discovery is to generate an operation that either *writes* outside of the destination buffer (an overflow), or *reads* outside of the source buffer (an overread). Additionally, vulnerability discovery ensures data leaked in an overread reaches an output function that is accessible by the attacker, like a socket *send*. This is also achieved using a goal-directed branch enforcement algorithm which ensures constraints are satisfied between the execution of the overread and the output function. When looking for a sink to exploit, Marten iterates through the sinks identified in the initial analysis to try to generate a vulnerability for each one.

2.2.3 Information Leakage Exploit. The information leakage exploit spills process RAM and reveals the randomized location of code pages. Marten parses the output of the program and identifies the target information it is trying to leak, in our current implementation the base address of `libc`. To avoid rerandomization associated

with restarting the process, Marten generates information leakage exploits that avoid memory corruption, specifically by ensuring that the destination buffer is large enough to hold the leaked data.

2.2.4 ROP Payload Generation. The final payload exploits a buffer overflow. To create this payload, Marten automatically finds the vulnerability and generates a ROP payload that meets the constraints of the target application so that the vulnerable target site can be reached. Marten supports the generation of multiple types of chains, each designed with bypassing modern defenses in mind. The first type of chain we implement, the *execve* chain, executes a *execve* syscall on a command or bash script supplied by the attacker. This script allows the attacker to execute arbitrary commands on the target system. The next chain type, the `shellcode` chain, uses the `mmap` syscall to map in a page that is both writable and executable. Then, it reads in code specified by the attacker to this page and executes it.

One challenge is ensuring the ROP payload is small enough to fit within a single TCP packet and can successfully navigate through the control flow of the program to reach the target. Marten therefore formulates a set of SMT constraints that capture the properties that the input must satisfy, then uses an SMT solver to solve the constraints and deliver the final attack input.

3 CASE STUDIES

Marten generates working exploits for four vulnerabilities across three code bases. These exploits have been verified to work against fully randomized, remotely accessed stripped binaries compiled in three different environments, specifically Ubuntu 18.04, Debian 9.9, and Red Hat Enterprise 8.0. All of our final payloads were generated on a virtual machine with 40 Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz processors and 500 GB of RAM running Ubuntu 18.04. All exploits generated in under six minutes on this machine. In general, the offsets of ROP gadgets within `libc` may be different for different versions of Linux. Marten therefore works with the appropriate version of `libc` when generating the ROP chain for processes running on the corresponding version of Linux. Figure 2 summarizes the applications, vulnerabilities, bypassed defenses, and platforms on which the final exploits were verified to work.

3.1 Dnsmasq

CVE-2017-14494: CVE-2017-14494 is an information leak vulnerability caused by a buffer overread. It is triggered when a malformed DHCPv6 forwarding packet is sent to the target which causes private information to be copied into a buffer. This information is then sent back to the attacker, leaking data to the attacker.

3.1.1 Trace Generation and Analysis. Marten compiles Dnsmasq-2.77 with our custom instrumentation. It sends a benign DHCPv6 forwarding packet (a provided benign input) to the instrumented version of Dnsmasq, which creates a trace file that contains information about operations performed on tainted data. Marten analyzes the trace file to discover vulnerable points in the program's execution. Examples of vulnerable points include calls to memory transfer functions (memcpy, memmove, recv) where the size of the transfer is influenced by the attacker. In this example Marten finds one potentially vulnerable point. Marten uses the information in the trace

```
1    void *put_opt6(void *data, size_t len)
2    {
3    void *p;
4
5    if ((p = expand(len)) && data)
6    ▷       memcpy(p, data, len);
7
8    return p;
9    }
```

CVE-2017-14494

```
"src_expr": (BitVec 64) bv548
"src_addr": 94435196822144
"index_expr": (BitVec 64) bv94435196822182
"size_expr": (BitVec 16)((byte_0x24<<8)+byte_0x25)
```

CVE-2017-14494 Symbolic Expressions

Figure 3: Vulnerable code and generated symbolic expressions for CVE-2017-14494

file to derive *symbolic expressions* for relevant function parameters such as the length passed to the memory transfer function. Each symbolic expression captures the complete computation that generated the corresponding value as a function of the input bytes and program constants. Marten also generates any *constraints* placed on each of the input bytes by branches that occur in Dnsmasq.

3.1.2 Vulnerability Discovery. Marten iterates over all of the target locations determined by the trace analyzer. Figure 3 presents the symbolic expressions for the values of parameters to the call to `memcpy` on line 6 of Figure 3 (we present the expressions in Z3 syntax [5]). The symbolic expressions show why the `memcpy` is vulnerable. The `src_expr` is the size of the buffer Marten is attacking when it was allocated. The `src_addr` is the concrete value of the source buffer's base address. The `index_expr` is the pointer into the buffer used as the source for the `memcpy` call. Finally, the `size_expr` is the symbolic expression which dictates the number of bytes copied in the `memcpy`.

The `size_expr` shows that the value of the size is determined by the 0x24th and 0x25th bytes in the input. There are no constraints set on these bytes by the program. There are 38 bytes between the start of the allocated buffer and the beginning of the source buffer used in the `memcpy`. This is determined by the difference between the `index_expr` and the `src_addr`. With these parameters, Marten generates a *symbolic equation* of the following form to generate an overread:

$$size\_expr > src\_expr - (index\_expr - src\_addr)$$

Marten uses Z3 [5] to solve the resulting symbolic equation to generate an overflow by setting the `size_expr` to a value exceeding 510. Address Sanitizer (ASAN) [23] then reports an error. Marten ensures that the data which is read out of bounds reaches an output source such as a socket send call. Marten makes this check so that it knows the data will be sent back. This turns a buffer overread error into an information leakage vulnerability.

3.1.3 Information Leakage. Marten now tailors the input to leak desired information. To do this, Marten runs an uninstrumented version of the application under GDB, using an automated interface to GDB to stop execution at the `memcpy` shown in line 7. It then

Program	Vulnerability Type	CVE	FULL ASLR	NX	RELRO	FORITIFY	Chains Generated	Verified On
Nginx-1.4.0	Buffer Overflow	2013-2028	✓	✓	✓	✓	`execve` `shellcode`	Ubuntu 18.04 Debian 9.9
OpenSSL-1.0.1	Buffer Overread	2014-0160	✓	✓	✓	✓		Red Hat Enterprise 8.0
Dnsmasq-2.77	Buffer Overflow	2017-14493	✓	✓	✓	✓	`execve` `shellcode`	Ubuntu 18.04 Debian 9.9
Dnsmasq-2.77	Buffer Overread	2017-14494	✓	✓	✓	✓		Red Hat Enterprise 8.0

Figure 2: Results for benchmark applications. Marten generates working exploits for four vulnerabilities and chains them together to create exploits which bypass full randomization and other defenses.

```
1    if ((opt = opt6_find(opts, end, OPTION6_CLIENT_MAC
         , 3)))
2    {
3    state->mac_type = opt6_uint(opt, 0, 2);
4    state->mac_len = opt6_len(opt) - 2;
5    ▷    memcpy(&state->mac[0], opt6_ptr(opt, 2),
         state->mac_len);
6    }
```

CVE-2017-14493

```
"dest_expr": (BitVec 64) bv16
"dest_addr": 140735078453168
"index_expr": (BitVec 64) bv140735078453168
"size_expr": (BitVec 16)((byte_0x24<<8)+byte_0x25)-2
```

CVE-2017-14493 Symbolic Expressions

Figure 4: Vulnerable code and generated symbolic expressions for CVE-2017-14493.

inspects the memory near the source buffer, trying to find the largest code page it can leak that is reachable by the `memcpy`. It determines that there is a pointer into `libc` which can be leaked, giving access to a huge number of potential ROP gadgets. Marten adjusts the length of the overread such that the size is greater than the distance between the start of the `memcpy` and the desired information. To ensure that the resulting exploit leaves Dnsmasq in an uncorrupted state, Marten also uses the symbolic expression for the length of the destination buffer to ensure that the leaked information fits in the destination buffer. In this example Marten computes the length of the overread as 0xFFFF — the length is stored in an unsigned short, with the maximum value of 0xFFFF less than the length of the destination buffer. At this point, Marten runs the application with this specially crafted input. It verifies that the desired target data is in the output and Dnsmasq continues to execute properly. Marten now has a successful information leakage exploit.

CVE-2017-14493 Marten next finds a buffer overflow vulnerability that enables remote code execution. Marten's vulnerability discovery algorithm discovers CVE-2017-14493, which is triggered when a malformed DHCPv6 packet is sent to the target that causes data to be written outside of a statically sized stack buffer.

3.1.4 Vulnerability Discovery. Line 5 of Figure 4 presents the vulnerable call to `memcpy`. The data in the source buffer and the number of bytes to copy is controlled by the attacker. To detect this vulnerability, Marten solves a symbolic equation over the variables in Figure 4. This equation takes the same form as the equation of CVE-2017-14494. Marten also ensures that it appends the proper amount of data to the malformed DHCPv6 packet such that there is enough to overflow the target buffer. These extra bytes are placeholder values that Marten replaces with a ROP chain in the next step. When the vulnerability is uncovered by a crash in ASAN, the ROP chain generation begins.

3.1.5 ROP Chain Generation. The ROP chain generation takes as an input the target library whose pointer is leaked in the information leakage step. In this case, it is the version of `libc` used by the target application. From here, the payload generation has access to a huge number of gadgets and can quickly synthesize both `shellcode` and `execve` ROP chains. Marten minimizes the length of the ROP chain by searching for gadgets that perform the desired computation with few (three or less) extraneous instructions. In practice, targeting `libc` gives Marten access to a wide enough range of gadgets that Marten is able to find exact matches for each desired gadget so that the final ROP chains have few extraneous instructions. The chain generation algorithm also takes special care to pad the generated payloads with offsets into the library which point to `ret` instructions. This is the ROP equivalent of a `NOP` sled, which is a technique to improve the robustness of an exploit against changes to the environment [13]. Then, the payload generation appends the attack code, which is also a set of offsets pointing to short code sequences in the leaked library. Marten generates a separate ROP payload file for each version of `libc` (one for each version of Linux) — the offsets of the gadgets are different for each version.

Figure 5 presents statistics for the generated ROP chains for our target libraries. The ROP chains contain either 62 (`execve` exploit) or 22 (`shellcode` exploit) gadgets. The generated chains contain four kinds of gadgets: write, load, syscall, and call depending on the action the gadget performs. The length of the ROP chains varies between 1496 and 422 bytes, with the amount of padding included in the ROP chain either 48 or 80 bytes. The padding consists of the address of a `ret` instruction in `libc`. To ensure that the chains fit within a single TCP packet, Marten generates ROP chains that are less than 1500 bytes.

The generate ROP payload is placed within the placeholder bytes appended to the malformed DHCPv6 packet in the vulnerability discovery step. From here, Marten solves any unsatisfied constraints created by replacing placeholder bytes by the ROP payload and adjusts the payload so that it exercises the target `memcpy` and executes the ROP payload.

Libc In	Chain Type	Gadget count	Instruction Count	Gadget Type[Number]	Length (bytes)	Padding (bytes)
Red Hat 8.0	execve	62	138	write[19], load[42] syscall[1]	1496	48
Red Hat 8.0	shellcode	22	42	write[5], load[15] syscall[1], call[1]	544	80
Debian 9.9	execve	62	62	write[19], load[42] syscall[1]	920	80
Debian 9.9	shellcode	22	22	write[5], load[15] syscall[1], call[1]	422	80
Ubuntu 18.04	execve	62	138	write[19], load[42] syscall[1]	1496	48
Ubuntu 18.04	shellcode	22	42	write[5], load[15] syscall[1], call[1]	544	80

Figure 5: Statistics of ROP chains generated for each target system's library.

3.2 Final Exploit

The final exploit is a Python script that receives a target IPv6 address, a payload file for the information leak, and a payload file for the stack buffer overflow complete with relative offsets for the ROP gadgets. Marten will generate a Python script using a template that sends the information leak payload to the server and receives the leaked data. The script will search for the target pointer using the offset it saved in the information leak payload generation step and determine the base address of `libc`. Once the script has found this base address, it will add it to all of the relative offsets in the ROP payload to generate absolute addresses. Then, this final payload file is sent to the target process. The result is an exploit which can perform arbitrary commands on the victim system that bypasses full ASLR and other modern defenses such as NX. We verified that the exploits work against Dnsmsaq-2.77 compiled on Ubuntu 18.04, Debian 9.9, and Red Hat Enterprise 8.0.

3.3 Nginx and OpenSSL

Marten begins with the source code for Nginx-1.4.0, OpenSSL-1.0.1, and two benign inputs to the program. The information leak vulnerability, CVE-2014-0160, is the infamous Heartbleed bug. Our seed input for the information leak is a benign `Client_Hello` and `Heartbeat` packet. First, Marten compiles this application with our LLVM based instrumentation. Marten runs the instrumented program and feeds it the benign packets, causing the process to log a trace of the program's execution. Our trace analyzer determines the points in the program that are potentially vulnerable. The analyzer logs data about these points including symbolic expressions representing the constraints on the input bytes that had to be satisfied to reach the target sites and an expression for the size of the vulnerable memory transfer. As opposed to the other traces which only find one potentially vulnerable point, Marten finds twenty-five points that are potentially vulnerable for Heartbleed. There are a greater number of potentially vulnerable points for this input because the the `Client_Hello` message contains data that gets used in hashing functions which have loops over the input data.

Marten enters its vulnerability discovery phase to generate an input which triggers a bug in the program. Marten iterates through all of the potentially vulnerable points determined by the trace analyzer. Eventually, Marten tries to attack the point which is truly vulnerable.

Marten mutates the input file such that the length of the heartbeat packet is longer than the source buffer. Marten uses the symbolic expressions previously logged to mutate the file. These expressions are fed into the Z3 [5] constraint solver that creates the required length while satisfying the constraints needed to reach the vulnerable code. Supplying this file to Nginx caused Address Sanitizer to report an error.

The bug triggering input is supplied to Marten's information leakage phase which runs against an uninstrumented version of the application. Marten's automated GDB based analysis creates an input which leaks an address in `libc`. Our library finding algorithm determines the location of the library address in the leaked data. From this address, Marten calculates the base address and created a ROP chain that is effective irregardless of how the target program is randomized by full ASLR.

Marten follows a similar approach to obtain a vulnerability triggering input for the buffer overflow bug. Marten is given a benign `chunked Transfer-Encoding request` packet. It then recognizes a signed-unsigned comparison in Nginx which results in an overflow vulnerability. This allows Marten to crash the target application.

Because of the library address leaked from memory, Marten has access to a huge number of ROP gadgets and synthesizes an exploit for CVE-2013-2028, the buffer overflow in Nginx. Our generated exploit's gadgets are based on offsets into the leaked library such that a new base address could be applied to them for any re-randomized run of the application. Marten automatically synthesizes ROP chains for the vulnerability and fits them into the symbolic equation for the exploiting input. Marten generates ROP payloads for each flavor of Linux it attacks, because their versions of `libc` are all slightly different. This causes their gadgets to be in different locations. Our exploits download and run a program from

a server. Marten verifies that the program is running on the victim system to ensure the exploit's success.

The final output of Marten is a Python script and payload files that will do both the information leakage attack and send a ROP payload to the target system. Marten verifies this script by using it against binaries compiled and running on remote systems using Ubuntu 18.04, Debian 9.9, and Red Hat 8.0. We verified that the script caused an arbitrary program to be downloaded and run by verifying that the injected process was executing on the target system.

4 PROGRAM INSTRUMENTATION

Marten takes the source code of an application and compiles it using an augmented version of Dataflow Sanitizer (DFSAN) [9]. This program is then run with a given input and Marten traces its execution while tracking taint as the input flows through the target application. Finally, we implement a trace analyzer for the logged information which tracks input bytes and their symbolic expressions.

4.1 Logging Execution Traces

Our instrumentation creates log files that track information about how the program computes and uses tainted values. Marten uses this information to derive symbolic expressions for the tainted values that the program computes and for the constraints placed on tainted values by the execution of conditional statements. Systems like Valgrind [11] do not log data and instead track data at runtime in memory. Our approach reduces the memory overhead of storing this data while running an instrumented application. Also, we minimize the impact that our instrumentation has on the target program's runtime by taking advantage of compiler optimizations to provide faster analysis.

4.2 Trace Analyzer

The Marten trace analyzer reads the log files to find `sinks` in the program which Marten will later attack. A sink is a function where data is used in a way that may cause an overflow, an overread, or perform transformations on the input bytes. Marten works with four different kinds of sinks:

- **Memtransfer:** A function corresponding to a memory transfer function, such as `memcpy`, that Marten is trying to attack. These are generated while Marten is looking for buffer overflows when the destination or size of a memory transfer is tainted.
- **Input:** A function which reads tainted data. These are generated any time data is read into the program from the attack input.
- **Overread:** A function corresponding to a memory transfer in which the source pointer or size is tainted. These are only generated when Marten is looking for buffer overreads.
- **Data:** A sink generated to represent tainted data that later flows into the other types of sinks. These are generated to support transformations of the data that flows into other sinks for both ROP chain generation and information leakage.

The trace analyzer will generate sinks depending on whether the parameters are tainted and if the sink type is of interest based on the type of attack Marten is implementing. When Marten attacks stack buffer overflows, it is interested in Memtransfer and Input sinks. In this kind of attack, Marten is targeting a vulnerable `memcpy` or `recv` in the program. When Marten generates information leak exploits, it is targeting overread sinks and data sinks.

Sinks include important information such as the constraints on the input bytes which allow us to reach a certain function of interest, the symbolic equations for the input, and concrete values for function parameters. Furthermore, Marten logs the taint labels of specific bytes so that it can determine where data that is manipulated in an overread is used again in an output function like a socket `send`.

Symbolic Expressions and Constraints: The trace analyzer keeps track of the symbolic expressions of individual bytes in the input. Marten tracks expressions for each of the bytes in the input. Once an input function, such as `read` or `recv`, is activated on a communication channel that is controlled by Marten, the trace analyzer will create symbolic variables for the bytes read in. Consider a symbolic variable called `byte` which is created when some input is read in.

From this point on, any calculation performed on information associated with these symbolic variables will propagate symbolic expressions to the relevant destinations. These calculations are checked by examining the LLVM bytecode generated at compile time. For example, say the LLVM register `i8` has the symbolic expression `byte`. If an LLVM instruction of the following form takes place :

```
%i9 = add i32 %i8, 10
```

Marten now knows that the register `i9` will have the symbolic expression `byte + 10`. If `i9` is later used to determine the size of a `memcpy` and Marten knows the size of the destination buffer, it will be able to determine if a concrete value exists for `byte` that will overflow the buffer.

Marten also keeps track of the constraints placed on the input bytes based on the branches that occur within the program. These constraints are represented with symbolic expressions that include a result of `True` or `False`. Marten uses the constraints extensively in vulnerability analysis.

5 VULNERABILITY DISCOVERY

After Marten has generated the relevant symbolic equations for targets sites triggered by the input, it manipulates the input into exposing a vulnerability. To do this, Marten follows the same methodology as the goal directed conditional branch enforcement algorithm described by Sidiroglou et al. [27]. We extend this technique by applying their method to buffer overflows and buffer overreads. Marten continuously runs the target application first with no constraints and then by adding constraints one by one until it detect a bug. To detect a bug, Marten recompiles the target application with Address Sanitizer (ASAN) [23] which will detect out of bounds reads and writes.

5.1 Vulnerability Discovery Algorithm

We implement an algorithm based on goal directed conditional branch enforcement to transform a benign input that traverses a

target site into an input which triggers a vulnerability. Algorithm 1 shows our approach to this problem.

Algorithm 1: Vulnerability Discovery Algorithm

```
input: C: source code of the program L: LLVM
  instrumented program S: the target sinks generated from
  initial analysis, I: program input
output: ρ: an input that triggers the vulnerability or ⊥
A ← ASAN_compile(C)
for s in S do
    cur_con_s ← ∅
    while cur_con_s ≠ all_con_s do
        if not ρ ← solver_generate_input(I, cur_con_s)
         then
            break
        end
        if bug_found(A(ρ)) then
            return ρ
        end
        mod_con_s ← get_constraints(L(ρ)) if
         got_new_constraint(cur_con_s, mod_con_s) then
            add_new_constraint(cur_con_s, mod_con_s)
        else if
         constraint_not_flipped(cur_con_s, mod_con_s)
         then
            flip_newest_constraint(cur_con_s)
        else
            break
    end
end
return ⊥
```

Given the source code of a program C, a version of C compiled with our *LLVM* instrumentation L, the sinks generated from initial analysis S, and the benign seed input I, the algorithm will first compile an ASAN version of the source code. Then, it will iterate through the sinks in S, trying to attack each of them.

Each sink has constraints associated with it. Marten starts by mutating I such that the new input triggers a vulnerability without adding any constraints. If the solver is unable to generate an input which meets the current constraints, Marten breaks and continue on to the next sink. If this input fails to trigger a bug, Marten will obtain the branches from the mutated input and add one new constraint associated with the target sink. If no new constraints have been generated, Marten will flip the newest constraint and continue executing. If no new constraints are generated and Marten has already tried flipping the newest constraint, Marten determines that it is not possible to generate an input which exercises a vulnerability for this sink. Thus, Marten moves on to the next sink and tries to attack it. Marten stops once it has found a bug or if it has tried all of the sinks in S and was unable to trigger a vulnerability.

5.2 Information Leakage Taint Tracking

Algorithm 1 targets both overread and overflow vulnerabilities. One major difference between overread and overflow vulnerabilities is that overreads violate the bounds of the the source buffer, while overflow bugs violate the bounds of the destination buffer. For a successful information leak, the data read outside of the source buffer must be sent back to Marten. Otherwise, this data will just reside in the application's RAM and be of little use.

To ensure the information from an overread reaches an output source which is attacker accessible, Marten reruns our initial analysis using the input which triggers the overread. This overread will apply taint to the data that Marten wants to read back. Marten tracks these taint labels as they flow through the program in the same way it tracks all other taint, using DFSAN. Once this tainted data reaches an output source that is accessible to Marten, it determines that it has a successful exploit. If this data does not reach an output source, Marten continues to solve constraints similarly to Algorithm 1 until the data is either read back or the constraints become unjustifiable.

Preventing Memory Corruption: Marten takes special care to ensure that it does not crash the target program after triggering the buffer overread. This goal is achieved by ensuring that the destination buffer is large enough to hold the data being read into it. Marten logs the size of the destination buffer in our sinks when it traces the process's execution and verifies the buffer is large enough to hold the data by checking its allocated size against the length of the overread.

6 INFORMATION LEAKAGE

To bypass full ASLR, Marten takes advantage of information leakage vulnerabilities. We provide a novel approach for exploiting these types of bugs. Once the bug is found, Marten crafts an input that leaks the information it wants. Marten leaks the addresses of code sections to bypass full ASLR. Marten implements a novel algorithm to discover target information by inspecting points of interest in the program and ensuring the desired data is leaked back to Marten. A scripted version of GDB is invoked by Marten to inspect the processes at runtime [28].

6.1 Information Discovery Algorithm

Algorithm 2 sets a breakpoint at a location of interest determined by the vulnerability discovery component of Marten. Next, it runs the program until it reaches the desired breakpoint. If this is the point at which the overread occurs, Marten inspects the program's state to determine the ranges for its executable memory pages, otherwise Marten will continue executing. When Marten reaches the overread, it attempts to find a pointer into the largest code segment located within reach of the overread and take note of its value. To find this pointer, Marten obtains the executable memory maps for the process from the respective `/proc/`*pid*`/maps` where *pid* matches that of the target process. This allows us to verify that the pointer is within an executable code segment. At this point, Marten mutates the input file to generate an overread which leaks the greatest amount of information it can obtain without overflowing the destination buffer.

6.2 Information Leakage Verification

Marten verifies that the information leakage exploit is successful by first running the target application and stopping execution on

Algorithm 2: Information Leakage Algorithm

input: C: source code of the program, S: the point in the program Marten is targeting, V: vulnerability triggering input
output: ϵ: leaking input or $\perp$
$P \leftarrow DEBUG\_compile(C)$
set_break(S, P)
$P_i \leftarrow$ **run** $P(V)$
while $break\_reached(P_i)$ **do**
  if $correct\_point(P_i, S)$ **then**
    $X \leftarrow executable\_segments(P_i)$
    $B \leftarrow get\_bits\_of\_X\_at\_point(P_i, X)$
    break
  else
    continue
  end
end
$\epsilon \leftarrow mutate\_input(V)$
if $verify\_output(\epsilon, P, B)$ **then**
  return ϵ
else
  return $\perp$
end

the first instruction after all libraries have been loaded. At this point, Marten inspects the memory map of the process and gathers the memory range for the library it intends to leak. After this, Marten lets the program run to completion. Marten sends the exploiting input to the program and verifies that there is a pointer in the response which resides somewhere in the target memory map. If this is the case, Marten knows that it has generated an input which successfully leaks critical information for the next stage of exploitation. Marten can now use all of the code in this section of the program to break full ASLR.

6.3 Information Identification

One challenge with automating information leakage is determining what is and what isn't a library address in the output. This challenge is compounded by the fact that Marten is finding information that is leaked from the heap. This memory section's layout is altered due to common operations like garbage collection, writing to log files, servicing requests from other clients, etc. The attacker has no control over these operations.

Marten uses a novel method for finding library addresses. It first generates the largest information leak it can without corrupting the target, causing the highest probability of the target data being leaked. It then takes advantage of the bits of entropy used to realize ASLR. On 64 bit Linux systems, libraries have 28 bits of entropy [10]. A library address takes on the following form for 64 bit systems:

```
0x00007fXXXXXXXYYY
```

In the above example, the bits represented with an `X` are randomized and the bits represented with a `Y` are the bottom 12 bits of an offset from the library's base address. Marten determines the value

```
1  ▸ 0x442067 <put_opt6+46> mov rdx, qword ptr [rbp - 0x20
       ]
2    0x44206b <put_opt6+50> mov rcx, qword ptr [rbp - 0x18]
3    0x44206f <put_opt6+54> mov rax, qword ptr [rbp - 8]
4    0x442073 <put_opt6+58> mov rsi, rcx
5    0x442076 <put_opt6+61> mov rdi, rax
6    0x442079 <put_opt6+64> call memcpy@plt <0x402a10>
```

Figure 6: Disassembly when setting breakpoint in outpacket.c at line 84 for Dnsmasq-2.77

of YYY when it selects a target to leak from automated analysis with GDB. From this point, finding the information is a matter of applying a bitmask to each 8 byte value in the output until Marten finds a value that matches the form of `0x00007fXXXXXXXYYY`. At this point, Marten subtracts the offset gathered from GDB analysis and can determine the base address of the target library.

Marten relies on the fact that standard libraries such as `libc`, unlike applications, are typically compiled once for each Lunix release and are then distributed along with the release. The code offsets within `libc` are therefore typically the same across all machines running the same Linux distribution so that the ROP chain generated for that version of `libc` will typically work across all of those machines.

6.4 GDB Scripting

We created a python class, `GDBWrapper`, which invokes GDB as a subprocess and provides various member functions that perform the useful functionality of different commands. We extend this functionality to create new commands not natively supported by GDB. For example, we implemented a member function `get_params_by_func` which will tell Marten the parameters to a function depending on what the current function being called is.

One of the challenges with automating GDB is ensuring that the commands sent generalize to more than one program. This is why the information we provide in our sinks from initial analysis is essential. From the LLVM bytecode, we are able to determine the source file and line of a specific function of interest. Once we have identified the overread sink we wish to attack, ensuring that we can analyze the interesting point of this program is as simple as setting a breakpoint at the proper location.

Breaking at a line number is a higher level of abstraction than breaking on a specific machine code instruction. As can be seen in Figure 6, when a breakpoint is set at the target location in the source code, GDB ends up stopping execution a number of instructions before the `memcpy` call. Thus, we ensure that we step for a few instructions until we are on the `call memcpy` opcode. This ensures the registers are properly set up to analyze the state of the program. Note that we may hit the breakpoint multiple times during the execution of the program. We therefore check the transfer size and stop only when the size matches.

Once we reach the location of interest, we have to inspect the program state to find the useful information. To do this, we rely on the fact that the `x86_64` calling convention places the arguments in the registers `RDI`, `RSI`, and `RDX`, respectively. Thus, we know that address of the source buffer in the `memcpy` of Figure 6 is in `RSI`. We can inspect memory located after this pointer to find the

information we wish to leak and generate an input file that causes this memory transfer to reach the target information.

7 EXPLOIT GENERATION

Once an information leak has been performed, the results are used to generate an exploit that bypasses full ASLR and other modern defenses. This is a three step process consisting of ROP chain generation, stack offset calculation, and exploit finalization. The final result is a Python script that takes an IP address for the target and automatically performs an information leakage attack and a buffer overflow attack which gives the attacker arbitrary computation and bypasses full ASLR.

Algorithm 3: execve chain generation

```
1  input: L: library for the target program, debug info not
     needed, O: Offset from base address of library, X:
     Command for the chain to execute
2  output: C: ROP chain that executes X
3  G = find_gadgets(L)
4  offset_gadgets(G,O)
5  D = get_data_segment_offset(C)
6  argv_addr = D
7  cmd_addr = D + length(X) + 0x8
8  null_address = cmd_address + length(X)
9  C += create_command(X, cmd_addr, arvg_addr)
10 C += write_memory(0x0, null_address)
11 C += create_dep_chain(%rax, %rdi, %rsi, %rdx)
12 C += syscall_gadget
13 C = add_padding(C)
14 return C
```

7.1 ROP Chain Generation

Our system builds on top of Ropper [21] to generate ROP [25] payloads, which are capable of bypassing NX and RELRO protections. We built upon the automatic ROP chain creation portion of Ropper, giving it the ability to generate exploits that are capable of much more flexible attacks than simply opening a local shell. Our current approach generates two different chains, the `execve` and `shellcode` chains.

execve: The `execve` ROPchain writes a command string to memory and then makes an `execve()` system call using that command string. The string may be an arbitrarily complex shell script. The only constraint is that the chain must be small enough to fit within the process's memory. The `execve` chain determines a place to write the string based on determining which locations in the library are both writable and readable. These are data sections used by the process to store information that is needed at runtime.

The `create_command()` function used during `execve` chain generation splits the provided string into architecture appropriate-sized pieces that are written by sequential write gadgets starting at `cmd_addr`, allowing for arbitrary length commands to be passed by users of Marten. Once the string literals have been written, an array of pointers to the strings are written to `argv_addr`.

The `create_dep_chain()` function takes in a set of register to value mappings and figures out an ordering of gadgets such that at the end of their execution, all of the desired registers will be set to the specified values without any of the registers being accidentally clobbered by previous gadgets. Algorithm 3 implements this technique.

shellcode: The `shellcode` ROPchain makes a `mprotect()` system call to make a portion of the data section executable, writes a small shellcode loader to the now executable memory, then jumps to the loader. The loader copies another shellcode to the addresses beneath it, then runs it.

As seen in the Algorithm 4, Marten first generates the chain M, which executes the `mprotect` system call on a desired memory range. Then, it generates a simple chain C which writes the loader to memory. Finally, the M chain is appended to C and returned. The C chain is generated second even though it executes first, as it requires the loader to be finished, which requires M to be finished so that Marten can check whether an additional pop is needed in the loader or not.

Algorithm 4: shellcode chain generation

```
1  input: L: library for the target program, debug info not
     needed, O: Offset from base address of library, X:
     Command for the chain to execute
2  output: C: ROP chain that executes C, SC : shellcode to be
     included in buffer overflow that will be written to an
     executable location
3  G = find_gadgets(L)
4  offset_gadgets(G,O)
5  SC = choose_shellcode(X)
6  D = get_data_segment_offset(C)
7  target_addr = D
8  target_addr_aligned = align(target_addr, 0x1000)
9  load_dst = target_addr + 0x100
10 SZ = 0x2000
11 P = 0x7 # (RWX permission)
12 M = create_dep_chain(%rax, %rdi, %rsi, %rdx)
13 M += syscall_gadget
14 jump_addr = target_addr + library_base_address
15 M += jump(jump_addr)
16 loader = build_loader(loader_dst, len(shellcode))
17 C = memwrite_chain(target_addr, loader)
18 C += M
19 C = add_padding(C) return C, SC
```

Information Leakage Integration: Marten specifies a base address from which gadget addresses are calculated. By feeding in the base address for the library targeted by information leakage, Marten is able to generate a ROP chain that uses the randomized offset that was leaked. This is done by using the gadgets located at specific offsets from the base address, stitching them into a chain, and applying the randomized base address once it is determined by our information leakage exploit.

7.2 Stack Offset Calculation

Once one or more ROP chains have been generated, Marten fits this chain into the vulnerability triggering input found in previous stages of the pipeline. To fit the chain, Marten requests a second input that contains a long De Bruijn sequence [1] that will overflow the buffer and overwrite the saved instruction pointer.

If Marten is able to generate the input that hijacks the return pointer of the program, the instruction pointer will be a portion of the De Bruijn sequence. This is used to quickly calculate the offset from the beginning of overflow to the saved instruction pointer.

If Marten was not able to generate such an input, but is able to find a smaller, non crashing input with specifiable fingerprint value, it will use our GDB wrapper to find the fingerprint value in memory. Once the fingerprint is found, Marten can calculate the difference between the location of the fingerprint and the saved instruction pointer.

7.3 Exploit Finalization

After one of the two stack offset calculation processes is run, Marten knows how many bytes need to be written to the buffer to cause a control flow hijack. For each generated ROP chain Marten can now ask the solver to assemble a final input that consists of whatever information is needed to get to the desired memory transfer, place arbitrary content to overflow the buffer, and then the values of the chain. If an input is able to be generated, Marten now has an attack file that can be sent to the target program which will bypass full ASLR and other modern defenses.

The final exploit is a Python script which takes the IP address of the target server and performs an information leakage exploit against it. The library base address that is leaked is applied to the relative offsets present in the attack file. The final attack file is then sent to the target which exploits the victim server and bypasses full ASLR.

7.4 Chain Size

Our chains are short for two primary reasons. First, information leakage gives Marten access to a huge number of gadgets because it has access to all of the code in large libraries like `libc`. This allows Marten to create chains using gadgets with few (three or less) extraneous instructions. For example, if Marten needs to set the `RDI` register, a gadget such as:

```
pop RDI; ret;
```

is typically readily available.

The second reason Marten is able to generate short chains is that it takes a heuristic approach to gadget searching. In Algorithm 3 and Algorithm 4, Marten searches for gadgets that meet very specific criteria. For example, when Marten wants to load a register it searches for gadgets that only have a single `pop` or `mov` instruction. This prevents Marten from searching for gadgets that have extraneous instructions which perform undesirable computation that is corrected with adding yet more gadgets to the chain. Our approach ensures Marten is building chains with gadgets that are compact. Marten generates chains that fit within a single TCP packet which is an important capability for generating reliable remote exploits.

7.5 Exploit Reliability

Our exploits are effective against stripped binaries that are compiled in different environments. This is due to three main reasons. First, our chains are short for the reasons listed in 7.4. They are short enough to fit within a single TCP packet. This prevent the chains from being split up across multiple packets and is an extremely important capability for remote exploits. If Marten is sending exploits to a remote server across a slow network, large chains may not reach the target server at the same time. If the server is using non-blocking receives, which is quite common, then this situation would cause only part of our chain to be loaded into memory. Because only part of the chain gets executed, our exploit would not finish and just crashes the target, an undesirable outcome for an attacker. Because Marten's chains fit into a TCP packet, they arrive at the sever as a unit and are loaded into memory all at once, enabling the exploits to complete their execution.

Marten also adds padding by augmenting its exploits with `ret` gadgets. This is the ROP equivalent of a NOP sled [13], a technique used to improve the reliability of exploits against unpredictable stack layouts. These extra gadgets at the beginning of our chains provide a buffer space to account for shifted stack configurations which may be due to different compilation settings of the target, different environment variables, etc.

Marten also gathers chains from libraries, not the program's code. This is significant, because programs may have different instruction sequences and offsets when compiled with different flag settings and optimization levels. From our own experience, compiling Nginx-1.4.0 in a different directory caused the gadgets in the binary to change. Libraries, especially `libc`, are typically compiled once, then distributed with the operating system release. Marten's exploits therefore target code in `libc` rather than code in the application.

8 RELATED WORK

Automatic Exploit Generation: Brumley et al. showed, given source code access, that it is possible to automatically convert a patch into an exploit [3]. Their notion of an exploit was an input that violates the safety checks created by the patch, not one that hijacks the control flow of the target program. The generated exploits target standard undefended (unrandomized with executable stack and heap) binaries.

Marten, in contrast, does not require patches (only source code access and a benign input that exercises potential attack targets) and generates fully functional exploits that bypass full ASLR (as well as other defenses such as NX) to inject and execute attacker code into remote processes accessible only via a network connection.

AEG and *Mayhem* use symbolic execution to automatically find stack buffer overflows and format string vulnerabilities while generating exploits that target these bugs [2, 4]. *Mayhem* does this without access to source code while *AEG* requires source code. The exploits generated by these systems only work with unrandomized binaries where all defenses such as ASLR and NX are disabled. Furthermore, they rely on binaries having the specific compiler and operating system combinations of their compiled target program.

Marten, in contrast, 1) uses goal-directed conditional branch enforcement to find heap and stack overread and overwrite vulnerabilities deep within the program, 2) leverages information leakage vulnerabilities to bypass full ASLR defenses, 3) generates compact ROP exploits that bypass NX layouts 4) generates final exploits that work against stripped binaries compiled using different compilers and operating systems with different stack and heap layouts

Automatic Data Oriented Exploits: *FLOWSTITCH* is a system that makes exploits through manipulating the data and not the control flow of a program [8]. Their system automatically performs data corruption attacks. These attacks result in privilege escalation attacks, sometimes allow for command execution on the target system, and never allow for code execution. Their exploits only work in the presence of a weak form of ASLR where only the libraries, stack, and heap are randomized but not the text and bss segments. *FLOWSTITCH* can perform information leakage attacks, but relies on data and code sections which are not protected by randomization and requires the attacker to specify the data that they want to leak.

Unlike data oriented exploits, Marten generates remote control flow hijacking exploits against fully randomized binaries that automatically allow the attacker to gain arbitrary code or command execution on the target server. This gives the attacker more flexibility their attack's consequences and allows them to hijack the execution of processes that are only accessible through remote vectors. Marten's exploits also work in the presence of full ASLR and do not require small portions of the process's code or data segments to be unrandomized. Marten can leak addresses from a victim process with full randomization enabled and identifies the target data to leak automatically without requiring any attacker interaction.

From POC to Exploitable: *Revery* is a system which takes crashing Proof of Concept (POC) inputs and determines whether they could lead to exploitable conditions or not when attacking unrandomized binaries [31]. This system was only tested on unrandomized CTF binaries which are extremely simplified, small programs designed for hacking competitions. This system stopped when it could write to an arbitrary address or gained control of the instruction pointer. *Revery* cannot perform information leakage attacks.

Marten is an end to end system which finishes the job of developing a fully functional exploit that is effective against stripped, fully randomized binaries protected by modern defenses. These binaries may be generated on other systems and do not need to be compiled by Marten. Additionally, Marten is able to take a benign input and convert it into a POC input that exposes a vulnerability using a goal directed conditional branch enforcement algorithm. Marten is tested against and succeeds in exploiting real applications, not simple toy programs. Marten performs information leakage exploits successfully.

Automatic Exploit Hardening: *Q* is a system that automatically hardens POC exploits against modern defenses by replacing shellcode with ROP chains [22]. Their system's exploits target the unrandomized sections of a program protected by a weak form of ASLR that does not randomize the location of the text and bss sections in a binary. The exploits generated by their system either call a library function without control of the arguments, or writes to an arbitrary address and then stops. *Q* can only use the gadgets available in the binary to generate its chains because it cannot perform information leakage exploits.

Marten does not require an existing POC exploit to develop its own exploits. Instead, Marten takes a benign input to the target program and creates a POC input that exercises a vulnerability automatically. Marten can exploit programs protected by full ASLR because of its ability to perform information leakage attacks. Additionally, Marten develops robust. reliable exploits that can achieve arbitrary code or command execution. Marten generates chains using libraries, so it has access to many more gadgets and can build shorter chains because of this. Marten it able to use these libraries because of its ability to perform information leakage exploits.

Automatic Buffer Overread Discovery: BORG automatically finds and triggers buffer overreads using symbolic execution and and static analysis [12]. BORG generates inputs that cause buffer overreads which do not leak data. Instead, they cause the target applications to crash. Borg does not generate control flow hijacking exploits. BORG does not try to generate inputs for programs with defenses such as full ASLR enabled.

Marten is able to automatically find and trigger buffer overreads. Marten extends this capability to ensure that the data in the overread is sent back to the attacker. From this information, Marten can generate control flow hijacking exploits that bypass full ASLR and other defenses.

Goal Directed Conditional Branch Enforcement: DIODE implements a goal-directed conditional branch enforcement algorithm [27]. DIODE uses the algorithm to find integer overflow vulnerabilities in programs. DIODE does not generate exploits of any kind after finding these vulnerabilities.

Marten extends the goal-directed conditional branch enforcement algorithm to automatically find and trigger buffer overflow and buffer overread vulnerabilities as well as integer overflows. Marten uses this capability to generate information leakage and control flow hijacking exploits that bypass full ASLR and other modern defenses.

Automatic ROP Chain Generation: A number of other systems have explored the automatic ROP chain generation problem [7, 15–17, 21, 30]. All of these systems are meant to be interactive and to help a user generate ROP chains. These systems do not generate complete ROP chains. Instead, they assist the attacker by helping them find useful gadgets in the target binary. These systems do not have the capability to integrate with information leakage exploits. These systems do not attempt to build chains that are robust against changes to the environment

Marten is totally automatic and does not require user interaction after it begins running. Marten's ROP chains are able to load registers for function arguments and finish the job by calling target functions. Marten's ROP chains are integrated with its ability to generate information leakage exploits to generate chains that bypass full ASLR and other modern defenses. Marten is capable to generating two types of chains. One that injects and executes arbitrary shellcode and one that calls the `execve` system call to execute arbitrary commands.

9 CONCLUSION

Address space layout randomization poses a significant obstacle to the successful exploitation of memory corruption vulnerabilities. By automatically discovering and exploiting information leakage vulnerabilities, Marten can successfully derandomize remote randomized target processes. The resulting information about the locations of remote code enables Marten to generate successful ROP chains tailored to the specific randomized addresses of the remote process. Coupled with Marten's ability to automatically discover buffer overflow vulnerabilities, these ROP chains enable Marten to deliver a fully automatic end to end system that combines multiple vulnerabilities to exploit remote randomized processes. Results from our case studies highlight the success of this approach.

ACKNOWLEDGMENTS

This research was supported by DARPA (Grant HR001118C0059).

REFERENCES

[1] Sharon Anderson, Alan T Bankier, Bart G Barrell, Maarten HL de Bruijn, Alan R Coulson, Jacques Drouin, Ian C Eperon, Donald P Nierlich, Bruce A Roe, Frederick Sanger, et al. 1981. Sequence and organization of the human mitochondrial genome. *Nature* 290, 5806 (1981), 457.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic exploit generation. (2011).

[3] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 143–157.

[4] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[6] Solar Designer. 1997. Non-Executable User Stack. (1997). https://www.openwall.com/linux/

[7] Andreas Follner, Alexandre Bartel, Hui Peng, Yu-Chen Chang, Kyriakos Ispoglou, Mathias Payer, and Eric Bodden. 2016. PSHAPE: automatically combining gadgets for arbitrary method execution. In *International Workshop on Security and Trust Management*. Springer, 212–228.

[8] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic generation of data-oriented exploits. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 177–192.

[9] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[10] Hector Marco-Gisbert and Ismael Ripoll-Ripoll. 2016. Exploiting Linux and PaX ASLR's weaknesses on 32-and 64-bit systems. *BlackHat Asia* (2016).

[11] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[12] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. 2015. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 87–97.

[13] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

[14] Hilarie Orman. 2003. The Morris worm: A fifteen-year perspective. *IEEE Security & Privacy* 99, 5 (2003), 35–43.

[15] PAKT. 2019. ropc. (2019). https://github.com/pakt/ropc

[16] Nguyen Anh Quynh. 2013. OptiROP: the art of hunting ROP gadgets. (2013).

[17] Rapid7. 2019. Msfrop. (2019). http://www.offensive-security.com/metasploit-unleashed/msfrop

[18] Redhat. 2012. Hardening ELF binaries using Relocation Read-Only (RELRO). (2012). https://access.redhat.com/blogs/766093/posts/1975793

[19] Redhat. 2012. Position Independent Executables (PIE). (2012). https://access.redhat.com/blogs/766093/posts/1975793

[20] Redhat. 2014. Enhance application security with FORTIFY_SOURCE. (2014). https://access.redhat.com/blogs/766093/posts/1976213

[21] Sascha Schirra. 2019. Ropper. (2019). https://github.com/sashs/Ropper

[22] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy.. In *USENIX Security Symposium*. 25–41.

[23] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[24] Fermin J Serna. 2012. The info leak era on software exploitation. *Black Hat USA* (2012).

[25] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *ACM conference on Computer and communications security*. New York,, 552–561.

[26] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 298–307.

[27] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. 2015. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *ACM Sigplan Notices*, Vol. 50. ACM, 473–486.

[28] Richard Stallman, Roland Pesch, Stan Shebs, et al. 2002. Debugging with GDB. *Free Software Foundation* 51 (2002), 02110–1301.

[29] PaX Team. 2001. PaX address space layout randomization (ASLR). (2001). https://pax.grsecurity.net/docs/aslr.txt

[30] A Wally. 2019. nrop. (2019). https://github.com/awailly/nrop

[31] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1914–1927.

[32] Matthew Warren and William Hutchinson. 2000. Cyber attacks against supply chain management systems: a short note. *International Journal of Physical Distribution & Logistics Management* 30, 7/8 (2000), 710–716.