Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University

Abstract—**We show that it is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary or source code, against services that restart after a crash. This makes it possible to hack proprietary closed-binary services, or open-source servers manually compiled and installed from source where the binary remains unknown to the attacker. Traditional techniques are usually paired against a particular binary and distribution where the hacker knows the location of useful gadgets for Return Oriented Programming (ROP). Our Blind ROP (BROP) attack instead remotely finds enough ROP gadgets to perform a `write` system call and transfers the vulnerable binary over the network, after which an exploit can be completed using known techniques. This is accomplished by leaking a single bit of information based on whether a process crashed or not when given a particular input string. BROP requires a stack vulnerability and a service that restarts after a crash. We implemented Braille, a fully automated exploit that yielded a shell in under 4,000 requests (20 minutes) against a contemporary nginx vulnerability, yaSSL + MySQL, and a toy proprietary server written by a colleague. The attack works against modern 64-bit Linux with address space layout randomization (ASLR), no-execute page protection (NX) and stack canaries.**

I. INTRODUCTION

Attackers have been highly successful in building exploits with varying degrees of information on the target. Open-source software is most within reach since attackers can audit the code to find vulnerabilities. Hacking closed-source software is also possible for more motivated attackers through the use of fuzz testing and reverse engineering. In an effort to understand an attacker's limits, we pose the following question: *is it possible for attackers to extend their reach and create exploits for proprietary services when neither the source nor binary code is available?* At first sight this goal may seem unattainable because today's exploits rely on having a copy of the target binary for use in Return Oriented Programming (ROP) [1]. ROP is necessary because, on modern systems, non-executable (NX) memory protection has largely prevented code injection attacks.

To answer this question we start with the simplest possible vulnerability: stack buffer overflows. Unfortunately these are still present today in popular software (*e.g.,* nginx CVE-2013-2028 [2]). One can only speculate that bugs such as these go unnoticed in proprietary software, where the source (and binary) has not been under the heavy scrutiny of the public and security specialists. However, it is certainly possible for an attacker to use fuzz testing to find potential bugs through known or reverse engineered service interfaces. Alternatively, attackers can target known vulnerabilities in popular open-source libraries (*e.g.,* SSL or a PNG parser) that may be used by proprietary services. The challenge is developing a methodology for exploiting these vulnerabilities when information about the target binary is limited.

One advantage attackers often have is that many servers restart their worker processes after a crash for robustness. Notable examples include Apache, nginx, Samba and OpenSSH. Wrapper scripts like `mysqld_safe.sh` or daemons like `systemd` provide this functionality even if it is not baked into the application. Load balancers are also increasingly common and often distribute connections to large numbers of identically configured hosts executing identical program binaries. Thus, there are many situations where an attacker has potentially infinite tries (until detected) to build an exploit.

We present a new attack, Blind Return Oriented Programming (BROP), that takes advantage of these situations to build exploits for proprietary services for which both the binary and source are unknown. The BROP attack assumes a server application with a stack vulnerability and one that is restarted after a crash. The attack works against modern 64-bit Linux with ASLR (Address Space Layout Randomization), non-executable (NX) memory, and stack canaries enabled. While this covers a large number of servers, we can not currently target Windows systems because we have yet to adapt the attack to the Windows ABI. The attack is enabled by two new techniques:

1) Generalized stack reading: this generalizes a known technique, used to leak canaries, to also leak saved return addresses in order to defeat ASLR on 64-bit even when Position Independent Executables (PIE) are used.
2) Blind ROP: this technique remotely locates ROP gadgets.

Both techniques share the idea of using a single stack vulnerability to leak information based on whether a server process crashes or not. The stack reading technique overwrites the stack byte-by-byte with possible guess values, until the correct one is found and the server does not crash, effectively reading (by overwriting) the stack. The Blind ROP attack remotely finds enough gadgets to perform the `write` system call, after which the server's binary can be transferred from memory to the attacker's socket. At this point, canaries, ASLR and NX have been defeated and the exploit can proceed using known techniques.

The BROP attack enables robust, general-purpose exploits for three new scenarios:

1) Hacking proprietary closed-binary services. One may notice a crash when using a remote service or discover one through remote fuzz testing.
2) Hacking a vulnerability in an open-source library thought to be used in a proprietary closed-binary service. A popular SSL library for example may have

a stack vulnerability and one may speculate that it is being used by a proprietary service.

3) Hacking an open-source server for which the binary is unknown. This applies to manually compiled installations or source-based distributions such as Gentoo.

We evaluate all three scenarios. Ideally, for the first scenario we would test our techniques against production services for which we hold no information about the software, but we are constrained for obvious legal reasons. To simulate such a scenario, we tested against a toy proprietary service a colleague of ours wrote for which we had no information about source, binary, or functionality. For the second scenario, we target a real vulnerability in the yaSSL library [3]. This library was used by MySQL in past and we use that as the host application. For the third scenario, we target a recent (2013) vulnerability in nginx [2] and write a generic exploit that does not depend on a particular binary. This is particularly useful as the exploit will work on any distribution and vulnerable nginx version without requiring an attacker to write a specific exploit for each distribution and version combination (as is done today).

We implemented a new security tool, Braille, that makes BROP attacks highly automated. Braille can yield a shell on a vulnerable server in approximately 4,000 requests, a process that completes in under 20 minutes and, in some situations, in just a few minutes. An attacker need only provide a function that constructs a request of a minimum length to crash the server and append a string provided by Braille. The function must also return a single bit based on whether the server crashes or not.

Our contributions are:

1) A technique to defeat ASLR on servers (generalized stack reading).
2) A technique to remotely find ROP gadgets (BROP) so that software can be attacked when the binary is unknown.
3) Braille: a tool that automatically constructs an exploit given input on how to trigger a stack overflow on a server.
4) The first (to our knowledge) public exploit for nginx's recent vulnerability, that is generic, 64-bit, and defeats (full/PIE) ASLR, canaries and NX.
5) Suggestions for defending against BROP attacks. In summary, ASLR must be applied to all executable segments (PIE) and re-randomization must occur after each crash (at odds with `fork`-only servers). Holding the binary from the attacker or purposefully altering it may not be an effective security countermeasure.

II. BRIEF HISTORY OF BUFFER OVERFLOWS

Buffer overflows are a classic vulnerability with a long history of exploits [4]. Conceptually, they are relatively easy to attack. For instance, a vulnerable program might read data from the network into a buffer. Then, assuming the program lacks sufficient bounds checks to limit the size of the incoming data, an attacker could overwrite memory beyond the end of the buffer. As a result, critical control-flow state, such as return addresses or function pointers, could be manipulated. Stack buffer overflows tend to be especially dangerous because return addresses are implicitly nearby in memory due to function

calling conventions. However, attacks that target buffers on the heap are also viable [5].

In the early days of stack buffer overflows, it was common for an attacker to include malicious code as part of the payload used to overflow the buffer. As a result, the attacker could simply set the return address to a known location on the stack and execute the instructions that were provided in the buffer. Such "code injection" attacks are no longer possible on contemporary machines because modern processors and operating systems now have the ability to mark data memory pages as non-executable (e.g., NX on x86). As a result, if an attacker tries to run code on the stack, it would only cause an exception.

An innovative technique, known as *return-oriented programming* (ROP) [1], was developed to defeat defenses based on non-executable memory. It works by linking together short code snippets already present in the program's address space. Such code snippets, called *gadgets*, can be combined to form arbitrary computation. As a result, attackers can use ROP to gain control of programs without any dependence on code injection. Simpler variations of ROP are sometimes possible. For example, with *return-to-libc* attacks, a high-level library function can be used as the return address. In particular, the `system()` function is useful for attackers because it can run arbitrary shell code with only a single argument [6]. These attacks were very effective on 32-bit systems where arguments were passed on the stack, already under control of the attacker. On 64-bit systems, arguments are passed in registers, so additional gadgets are needed to populate registers.

Address space layout randomization (ASLR) [7], [8] was introduced as an additional defense against buffer overflow attacks. It works by randomizing the location of code and data memory segments in the process address space. In many implementations code segment randomization is only applied to libraries, but full address space randomization is also possible. ASLR creates a major challenge for attackers because it makes the address locations of code (or even the stack) impossible to predict in advance. Unfortunately, on 32-bit platforms, ASLR is constrained by the number of available bits (usually 16) for randomization. As a result, brute-force attacks can be quite effective [9]. However, on 64-bit platforms there are typically too many random bits for brute-forcing to be feasible. In such cases, ASLR can still be circumvented, but only when combined with a vulnerability that leaks information about the address space layout, such as a format string [10].

In addition to the larger address space for ASLR and the need to locate additional gadgets to fill argument registers, 64-bit systems present a third complication for attackers. Because the architecture limits virtual addresses to 48-bits, user-level memory pointers are required to contain zero-valued bytes. These zeros cause early termination of overflows relying on string operations such as `strcpy()`.

Canaries [11] are another common defense against buffer overflow attacks. Canaries cannot prevent buffer overflows, but they can detect them retroactively and terminate the program before an attacker can influence control flow. For example, with stack canaries, a secret value that was determined in advance is placed just before each saved frame pointer and return address. Then, when a function returns, the secret

```
dup2(s, 0);
dup2(s, 1);
dup2(s, 2);
execve("/bin/sh", 0, 0);
```

Figure 1. Socket reuse shellcode. It redirects stdin, stdout and stderr to the socket, and executes a shell.
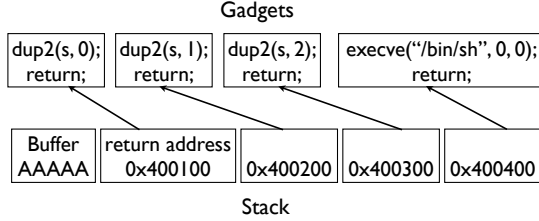

Figure 2. ROP version of the socket reuse shellcode. Gadgets are chained by placing their addresses on the stack.

value is checked to make sure it has not changed. This can prevent stack buffer overflows from being exploited because an attacker must correctly overwrite the secret value in order for the program to actually use an overwritten return address. However, just as with ASLR, canaries can be defeated through an additional vulnerability that leaks information about the secret value. The layout of stack memory can be an important consideration for canary implementations. One common approach is to place all buffers at the top of the frame so that if they overflow it will not be possible to overwrite other variables before corrupting the canary [12]. The motivation is to protect pointers because sometimes they can be used to overwrite arbitrary memory [13]. Unfortunately, canaries are not a perfect solution, as even with layout precautions, the structure of a buffer overflow can sometimes permit an attacker to bypass canary words and access critical state directly, as happened with unsafe pointer arithmetic in yaSSL [3].

III. ROP Tutorial

Before discussing the Blind ROP technique, we first familiarize the reader with ROP. Modern exploits rely heavily on ROP. The goal of ROP is to build an instruction sequence that typically spawns a shell (shellcode) based on existing code fragments (gadgets). Once a shell is executed, the attacker can execute more commands to continue the attack. Traditionally, exploits would inject off-the-shelf shellcode into the process and execute it. Figure 1 shows typical shellcode that pipes the attacker's socket to standard input, output and error and executes a shell.

Of course injecting shellcode is no longer possible because these days writable memory (*e.g.,* the stack) is non-executable, and so ROP must be used instead. Figure 2 shows how ROP can in principle be used to create the shellcode previously shown in Figure 1. The stack is overflowed so that the addresses of all the gadgets are present in sequence. Each gadget ends with a return so that the next gadget can execute.

In practice, each ROP gadget will be a short sequence of machine instructions terminated by a return. Executing a
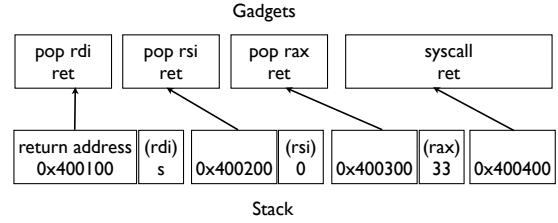

Figure 3. ROP chain for dup2(s, 0). The system call number needs to be in rax and dup2 is system call #33. Arguments are passed in rdi and rsi. Because the attacker already controls the stack, pop gadgets are used to load registers from values on the stack.

simple system call like dup2 will require multiple gadgets because arguments are passed in registers so gadgets to populate these will be needed. Figure 3 shows the required gadgets for dup2. Registers rdi and rsi control the first two arguments to system calls, and rax controls the system call number. Registers can be controlled by using pop gadgets and placing the value to load on the stack. By chaining enough gadgets, complete shellcode can eventually be built.

IV. Buffer Overflows today

On most contemporary operating systems, where NX and ASLR are common, an attacker must fulfill at least two requirements in order to gain full control of a remote program's execution:

1) To defeat NX, the attacker must know where gadgets reside inside the program executable.
2) To defeat ASLR, the attacker must derandomize the location at which the executable's text segment is actually loaded in memory.

These requirements can easily be brute-forced on 32-bit systems [9], [14] through simple guessing. This is not practical for 64-bit systems; in fact, most public exploits target 32-bit systems only. The purpose of the BROP attack is to circumvent these requirements on 64-bit systems. Hence, the rest of this discussion exclusively considers 64-bit attacks.

The first requirement in practice means that the attacker must have a copy of the vulnerable binary to disassemble and find gadgets. To our knowledge, our proposed BROP attack is the first general-purpose technique that can be used to defeat NX when the binary code is completely unavailable.

Defeating ASLR is also a significant challenge without BROP, but there are some possible strategies. Firstly, an information leak might reveal the address location of a code segment. Secondly, it may be possible to exploit any code that remains statically positioned across executions. For example, on Linux it is usually the case that the executable's code is mapped to a fixed address even though dynamic libraries and other data memory regions are randomized with ASLR. As a result, an attacker could simply apply ROP to the program's text segment directly. Additionally, on some platforms, such as Windows, there are shared libraries that are incompatible with ASLR, and thus such libraries are mapped to static locations.

On Linux, it is possible to apply ASLR to the entire address space, including the program's text segment, by enabling PIE.
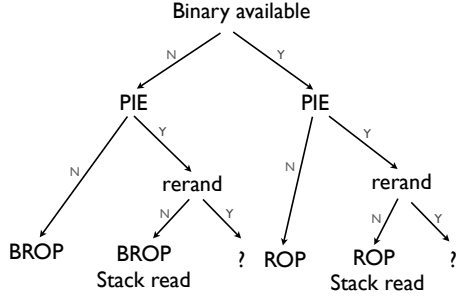
Figure 4. Techniques needed to attack different 64-bit scenarios. BROP and stack read are our contributions.

With GCC, this is achieved through the `-pie` flag. Using PIE has been recommended in previous studies [15], but unfortunately, it has not been widely deployed to date. When PIE is enabled, there are no known general-purpose 64-bit techniques, outside of our proposed generalized stack reading attack, that can be used to defeat ASLR.

Figure 4 shows how our BROP attack improves the state of the art in 64-bit exploit techniques. Today there are general techniques (ROP) to attack 64-bit servers only when the exact binary is available to the attacker and PIE is not used. Our stack reading technique makes it possible to attack PIE servers that do not rerandomize after a crash (*i.e.,* `fork`-only without `execve`). The BROP attack additionally opens up the possibility of hacking systems where the binary is unknown. In all cases, the BROP attack cannot target PIE servers that rerandomize (*e.g.,* `execve`) after a crash.

Hacking without binary knowledge is useful even in the not-completely-blind case (*e.g.,* open-source) because it makes it possible to write generic, robust exploits that work against all distributions and are agnostic to a specific version of the binary. Today, attackers need to gather exact information (*e.g.,* binaries) for all possible combinations of distribution versions and vulnerable software versions, and build an exploit for each. One might assume attackers would only bother with the most popular combinations. An implication of our work is that more obscure distributions offer little protection (through obscurity) against buffer overflows.

V. BROP ENVIRONMENT

The Blind Remote Oriented Programming (BROP) attack makes the following assumptions and requires the following environment:

- A stack vulnerability and knowledge of how to trigger it.

- A server application that restarts after a crash.

The threat model for a BROP attack is an attacker that knows an input string that crashes a server due to a stack overflow bug. The attacker must be able to overwrite a variable length of bytes including a return instruction pointer. The attacker need not know the source or binary of the server. The attacker is able to crash the server as many times as he wishes while conducting the attack, and the server must restart.
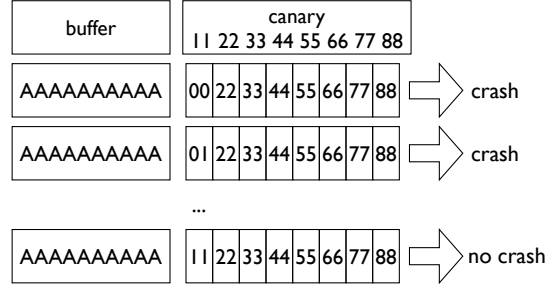

Figure 5. Stack reading. A single byte on the stack is overwritten with guess X. If the service crashes, the wrong value was guessed. Otherwise, the stack is overwritten with the same value and no crash occurs. After at most 256 attempts, the correct value will be guessed. The process is then repeated for subsequent bytes on the stack.

If the server is compiled with the PIE flag, the server must be a `fork`ing daemon and must restart without using `execve`. The same is true for overflows where the canary must be modified by the exploit. The attacker is also able to distinguish when a server crashes prematurely, *e.g.,* by noticing that the socket closes without receiving a response.

VI. ATTACK OUTLINE

The BROP attack has the following phases:

1) Stack reading: read the stack to leak canaries and a return address to defeat ASLR.
2) Blind ROP: find enough gadgets to invoke `write` and control its arguments.
3) Build the exploit: dump enough of the binary to find enough gadgets to build a shellcode, and launch the final exploit.

The first phase is needed so that a starting point address for scanning gadgets is found. Gadgets are then searched for until enough are found to invoke `write`. After that, the binary is transferred over the network from memory, enabling known techniques to be applied toward building the final exploit.

VII. STACK READING: ASLR DE-RANDOMIZATION

Exploits must have a method of defeating ASLR for configurations where PIE is used. We present a new stack reading technique that generalizes a known technique used for leaking canaries. It is useful even in cases where the binary is known and a full BROP attack is not required. The basic idea in leaking canaries is to overflow a single byte, overwriting a single byte of the canary with value x. If x was correct, the server does not crash. The algorithm is repeated for all possible 256 byte values until it is found (128 tries on average). The attack continues for the next byte until all 8 canary bytes (on 64-bit) are leaked. Figure 5 illustrates the attack. We generalize the attack to leak more words from the stack ("stack reading"). After the canary, one typically finds the saved frame pointer and then the saved return address, so three words need to be read. Figure 6 shows a typical stack layout.

There are a few subtleties that apply to generalized stack reading but not to reading canaries. With canaries, exact values

Figure 6. Typical stack layout. The canary protects saved registers. If it is overwritten (by an overflow) then the program aborts prior to returning to the saved return address.

TABLE I. AVERAGE REQUESTS NEEDED TO BRUTE-FORCE ASLR VERSUS OUR STACK READING TECHNIQUE.

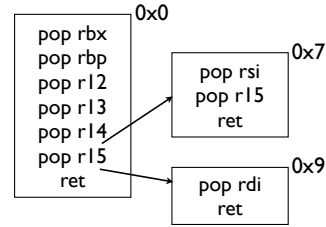| Platform	Entropy	Brute Force	Stack Reading
32-bit Linux	16-bits	2^{15}	512
64-bit Linux	28-bits	2^{27}	640
64-bit Mac OS X	16-bits	2^{15}	640

Figure 7. The BROP gadget. If parsed at offset 0x7, it yields a `pop rsi` gadget, and at offset 0x9, it yields a `pop rdi` gadget. These two gadgets control the first two arguments to calls. By finding a single gadget (the BROP gadget) one actually finds two useful gadgets.

will always be returned because there is only one value canary that is correct. In general though, stack reading will not necessarily return the exact saved instruction pointer present on the stack. It is possible that a slightly different value is returned depending on whether another value still resumes program execution without causing a crash. For example, suppose that 0x400010 was stored on the stack and the value 0x400007 is currently being tested. It is possible that the program keeps executing without crashing and 0x400007 is obtained from stack reading. This is OK as the attacker is searching for any valid value in the .text segment range and not for a specific one.

It is possible that stack reading does not return an address in the application's own .text segment, but rather a return address in a library. This can happen, for example, when the vulnerability lies in a library, or a callback happens. This is fine because gadgets can be found in the library instead. One can also stack read further to find more return addresses, if needed.

On 64-bit x86 systems only a portion of the address space is made available to the operating system (canonical form addresses). This allows us to skip several bytes when reading pointers. For user space processes the top two bytes are always zero. In fact, on Linux the third byte is `0x7f` for libraries and the stack. The main binary and heap are usually stored at `0x00` for executables compiled without the PIE flag. Thus we can skip on average three bytes (384 requests) when reading addresses.

Table I shows the complexity of using stack reading versus standard brute-force attacks. We compare 32-bit and 64-bit systems across several operating systems. Clearly the brute-force attack on 64-bit Linux is not practical and attackers have resorted to other techniques to circumvent ASLR. Many attacks have depended on non-randomized (without PIE) binaries that are common on Linux. Similarly Windows exploits have also resorted to attacking binaries that have opted out of randomization, or libraries that randomize once per reboot. Other attacks have used leaked pointers sometimes requiring another vulnerability.

The fact that stack reading succeeds tells the attacker that the BROP environment exists and that a stack overflow, rather than some random bug, is being triggered. A bug like a null pointer dereference may cause a crash for all possible byte values being probed, or a no crash for multiple possible values

(as opposed to one only). The words returned by stack reading give further evidence of the BROP attack working because the values can be somewhat sanitized: e.g., a random canary (which always starts with zero on Linux), a frame pointer, and a return address with known upper bits (0x40 for non-PIE or 0x7f).

VIII. BROP ATTACK

The BROP attack allows writing exploits without possessing the target binary. It introduces techniques to find ROP gadgets remotely and optimizations to make the attack practical.

A. The pieces of the puzzle

The goal is to find enough gadgets to invoke `write`. After that, the binary can be dumped from memory to the network to find more gadgets. The `write` system call takes three arguments: a socket, a buffer and a length. Arguments are passed in `rdi`, `rsi` and `rdx` registers, and the system call number is stored in the `rax` register. The following gadgets are therefore needed:

1) `pop rdi; ret` (socket)
2) `pop rsi; ret` (buffer)
3) `pop rdx; ret` (length)
4) `pop rax; ret` (write syscall number)
5) `syscall`

While an attack that finds all these gadgets is possible (see Section VIII-I) we first describe an optimized version that makes the attack more practical.

The first optimization is the *BROP gadget*. Shown in Figure 7, the BROP gadget is very common as it restores all callee saved registers. Misaligned parses of it yield a `pop rdi` and `pop rsi`. So by finding a single gadget, we find two gadgets that control the first two arguments of a call.

The second optimization is finding a `call write`. Instead of finding two gadgets (`pop rax; ret` and `syscall`) we can find a single `call write` instruction. One convenient place to find `call write` is the program's Procedure Linking Table (PLT). The PLT is a jump table used for dynamic linking containing all external library calls made by the application. Figure 8 shows the structure of an ELF binary; the PLT is the first region to contain valid executable code.
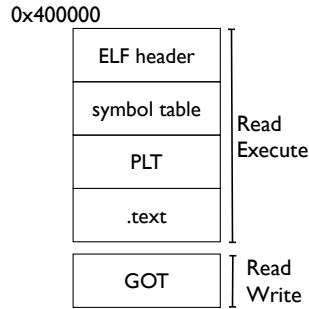
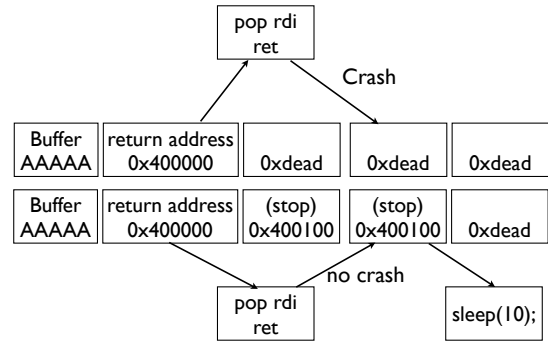Figure 8. ELF loaded in memory. The PLT contains a jump table to external functions (*e.g.,* libc calls).

Figure 9. Scanning for gadgets and the use of stop gadgets. To scan for gadgets one overwrites the return address with a candidate .text address (*e.g.,* 0x400000). If a gadget is found, it will return, so one must add "stop gadgets" to the stack to stop the ROP chain execution so that a return does not cause a crash, making it possible to detect gadgets.

The problem is now reduced to finding the BROP gadget, write's entry in the PLT and a way to control `rdx` for the length of the write. Any greater than zero length for write will do as one can leak the binary in multiple small chunks by chaining writes. We note that at the time of exploit, `rdx` may have a sane (greater than zero) value so having to control `rdx` may be unnecessary, but for the general case it is. Unfortunately `pop rdx; ret` gadgets are rare, so an optimization is to find a call to `strcmp` instead (again in the PLT) which sets `rdx` to the length of the string being compared. The optimized attack therefore requires:

1) Finding the BROP gadget.
2) Finding the PLT.
   - Finding the entry for `write`.
   - Finding the entry for `strcmp`.

B. Finding gadgets and the stop gadget

The basic idea in finding gadgets remotely is to scan the application's text segment by overwriting the saved return address with an address pointing to text and inspecting program behavior. A starting address can be found from the initial stack reading phase or 0x400000 can be used on default non-PIE Linux. Generally speaking two things will occur: the program will crash or it will hang, and in turn the connection will close or stay open. Most of the time the program will crash, but when it does not, a gadget is found. For example, 0x400000 may point to code with a null pointer dereference and cause a crash. The next address, 0x400001, may point to code which causes an infinite loop and keeps the connection open. These latter gadgets that stop program execution are fundamental to finding other gadgets: we call these stop gadgets.

A problem with using this technique naively for finding gadgets is that even if the return address is overwritten with the address of a useful gadget like `pop rdi; ret`, the application will still likely crash because it will eventually attempt to return to the next word on the stack, likely an invalid address. The crash would cause us to discard the gadget classifying it as uninteresting. Figure 9 shows this. To find gadgets we need to stop the ROP chain execution. This is where stop gadgets come in. A stop gadget is anything that would cause the program to block, like an infinite loop or a blocking system call (like sleep). To scan for useful gadgets, one places the address being probed in the return address,

followed by a number of stop gadgets. Note that a `pop rdi; ret` gadget would pop the next item from the stack into `rdi` so two stop gadgets would be needed in this case. Each time a useful gadget that does not cause a program crash is found, the stop gadget will run, blocking the program and leaving the socket open (instead of causing a crash). One can now scan the entire .text segment to compile a list of gadgets. The next section describes how the attacker can identify the instructions of a gadget—*e.g.,* differentiate between `pop rdi; ret` and `pop rsi; ret`.

Stop gadgets need not necessarily be gadgets that "stop" the program. They are merely a signaling mechanism. For example, a stop gadget could be one that forces a particular write to the network so the attacker can tell whether the stop gadget executed. Another scenario is one in which a stop gadget is already present in the stack frame. The stack will indeed already have multiple return addresses in it, and one of them may act as a stop gadget. For example a server may handle requests in a while-true loop, so returning to that loop may "resume" program execution and another request can be handled. This can be used to signal whether a program crashed or is still alive (*i.e.,* the stop gadget ran). The attacker in this case would populate the stack with the addresses of enough `ret` instructions to "eat up" enough stack until the next word on the stack is a return address of a previous stack frame that acts as a stop gadget (*e.g.,* returns to the main program loop). This particular optimization is useful for preventing situations where worker processes are limited and infinite loop-type stop gadgets cause all workers to become stuck, making it possible to continue the attack (as is the case with nginx). Section VIII-J describes in more detail how one can attack systems with few worker processes.

C. Identifying gadgets

We now discuss how to classify gadgets. This can be done by controlling the stack layout and inspecting program behavior. We define three values that the attacker can place on the stack:

Probe   The address of the gadget being scanned.
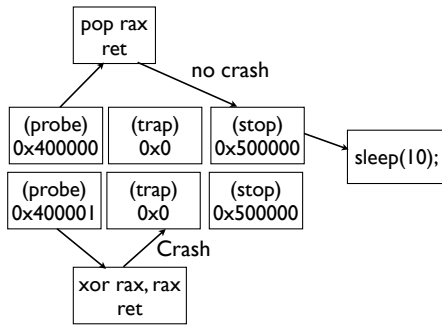Stop   The address of a stop gadget that will not crash.

Figure 10. Scanning for pop gadgets. By changing the stack layout, one can fingerprint gadgets that pop words from the stack. For example, if a "trap gadget" is executed rather than popped, the program will crash.
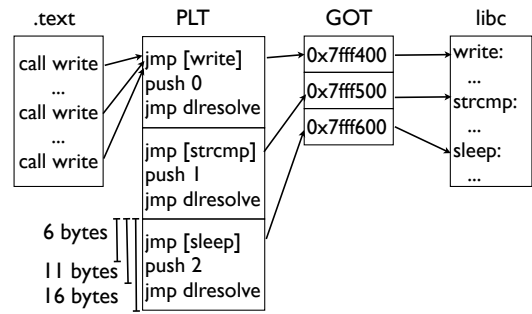

Figure 11. PLT structure and operation. All external calls in a binary go through the PLT. The PLT dereferences the GOT (populated by the dynamic linker) to find the final address to use in a library.

Trap   The address of non-executable memory that will cause a crash (*e.g.,* 0x0).

The idea is that by varying the position of the stop and trap on the stack, one can deduce the instructions being executed by the gadget, either because the trap or stop will execute, causing a crash or no crash respectively. Here are some examples and possible stack layouts:

- probe, stop, traps (trap, trap, . . .). Will find gadgets that do not pop the stack like `ret` or `xor rax, rax; ret`.

- probe, trap, stop, traps. Will find gadgets that pop exactly one stack word like `pop rax; ret` or `pop rdi; ret`. Figure 10 shows an illustration of this.

- probe, stop, stop, stop, stop, stop, stop, traps. Will find gadgets that pop up to six words (*e.g.,* the BROP gadget).

The traps at the end of each sequence ensure that if a gadget skips over the stop gadgets, a crash will occur. In practice only a few traps (if any) will be necessary because the stack will likely already contain values (*e.g.,* strings, integers) that will cause crashes when interpreted as return addresses.

By using the second stack layout one can build a list of pop x gadgets. One still does not know whether a `pop rdi` or `pop rsi` was found. At this point the attack diverges: one can either conduct a "first principles" attack that identifies pop gadgets based on system call behavior, or an optimized version of the attack that relies on the BROP gadget.

The BROP gadget has a very unique signature. It pops six items from the stack and landing in other parts of it pops fewer items from the stack so one can verify a candidate by laying out traps and stop gadgets in different combinations and checking behavior. A misaligned parse in the middle yields a `pop rsp` which will cause a crash and can be used to verify the gadget and further eliminate false positives. The gadget is 11 bytes long so one can skip up to 7 bytes when scanning the .text segment to find it more efficiently, landing somewhere in the middle of it. If more than 7 bytes are skipped one risks landing on `pop rsp` and thus not finding a copy of the gadget. After the BROP gadget is found, the attacker can control the first two arguments (`rdi` and `rsi`) to any call.

D. Finding the Procedure Linking Table

To control the third argument (`rdx`) one needs to find a call to `strcmp`, which sets `rdx` to the length of the string compared. The PLT is a jump table at the beginning of the executable used for all external calls (*e.g.,* libc). For example, a call to `strcmp` will actually be a call to the PLT. The PLT will then dereference the Global Offset Table (GOT) and jump to the address stored in it. The GOT will be populated by the dynamic loader with the addresses of library calls depending on where in memory the library got loaded. The GOT is populated lazily, so the first time each PLT entry is called, it will take a slow path via `dlresolve` to resolve the symbol location and populate the GOT entry for the next time. The structure of the PLT is shown in Figure 11. It has a very unique signature: each entry is 16 bytes apart (16 bytes aligned) and the slow path for each entry can be run at an offset of 6 bytes.

Most of the PLT entries will not cause a crash regardless of arguments because they are system calls that return EFAULT on invalid parameters. One can therefore find the PLT with great confidence if a couple of addresses 16 bytes apart do not cause a crash, and can verify that the same addresses plus six do not cause a crash. These addresses are also the first to have valid code as they are early on in the executable's address space.

The PLT can therefore be found by scanning from the program's origin (0x400000) or backwards from the address leaked through stack reading if the PIE flag was used. Each address must be 16 bytes aligned and 16 bytes can be skipped per probe for efficiency. We note that PLTs are often pretty large (200 entries) so one can skip even more bytes (thus skipping PLT entries) when looking for it to optimize for speed, hoping that a function that will not crash will still be hit.

The stack layout to find a PLT entry will be: probe, stop, trap. The PLT can then be verified by seeing if neighboring entries do not crash, and if offsets of six bytes (the PLT slowpath) still do not cause a crash.

E. Controlling rdx via strcmp

Once the attacker finds the PLT, the question is what function calls do various entries correspond to? One of them

will be `strcmp`, unless the program or one of its libraries does not use that function in which case the attacker can perform a "first principles" attack described later to find `pop rdx; ret`. We note that there is nothing particular to `strcmp` apart from it being commonly used and it setting `rdx` to a greater than zero value. Any other function that does the same will work.

The attacker can identify PLT entries by exercising each entry with different arguments and seeing how the function performs. The first two arguments can be controlled thanks to the BROP gadget. `strcmp` for example has the following behavior and signature, where "bad" is an invalid memory location (*e.g.,* 0x0) and "readable" is a readable pointer (*e.g.,* an address in .text):

- strcmp(bad, bad): crash

- strcmp(bad, readable): crash

- strcmp(readable, bad): crash

- strcmp(readable, readable): no crash

The attacker finds `strcmp` by finding an entry that responds to the previously mentioned signature. The PLT can be scanned in two ways. The attacker will have the address of a valid PLT entry, found previously. The naive technique is to probe addresses ± 0x10 bytes. A more effective technique that avoids running off either end of the PLT is using the PLT's slow path. The PLT slow path pushes the PLT entry number on the stack and then calls `dlresolve`. This call is present in each PLT entry at offset 0xb. One can therefore overwrite the saved return address with the PLT entry found + 0xb, followed by the entry number one wishes to probe (starting at zero) to systematically scan all PLT entries from the first one.

Once `strcmp` is found, the attacker can set `rdx` to a non-zero value by just supplying a pointer to either a PLT entry (non-zero code sequence) or the start of the ELF header (0x400000) which has seven non-zero bytes.

False positives can be found when searching for `strcmp`. Interestingly though, in all our tests we found that `strncmp` and `strcasecmp` were found instead, and all had the effect of setting `rdx` to a value greater than zero, thereby fulfilling the same purpose.

F. Finding write

The attacker so far can control the first three arguments to any call: the first two via the BROP gadget, and the third one indirectly via `strcmp`. `write` can now trivially be found by scanning each PLT entry and forcing a write to the socket and checking whether the write occurred. The only complication is figuring out the file descriptor number for the socket. There are two approaches: chaining multiple writes each with different file descriptor numbers in a single ROP chain, or opening multiple connections and using a relatively high file descriptor number in hope that it will match one of the connections. We use both techniques in combination.

Linux restricts processes to a maximum of 1024 simultaneously open file descriptors by default, making the search space small. Further, POSIX requires that new file descriptors use the lowest number available, so in practice searching the first few file descriptors works well.

G. Concluding the attack

At this point the attacker can write the entire .text segment from memory to the attacker's socket, disassemble it, and find more gadgets. The attacker can also dump the symbol table and find useful functions in the PLT like `dup2` and `execve`. Generally speaking the attacker will need to:

1) Redirect the socket to standard input / output. The attacker can use `dup2` or `close`, followed by either `dup` or `fcntl(F_DUPFD)`. These are often in the PLT.
2) Find "/bin/sh" in memory. An effective technique is to find a writable memory region like the environment, `environ`, from the symbol table, and `read` "/bin/sh" from the attacker's socket to that address.
3) `execve` the shell. If `execve` is not in the PLT, the attacker will need to transfer more of the binary to find a `pop rax; ret` and `syscall` gadget.

Dumping the symbol table is not as straightforward as one might hope. While the ELF header is loaded in memory, the section table (at the end of the binary) is not. The section header contains information about the start of the symbol table. In order to find the symbol table without this information, the attacker must start dumping the binary from the start until ASCII strings (function names) are found (which is the dynstr section). Based on the dynstr section, other adjacent sections containing symbol table information can be found.

H. Attack summary

The optimized BROP attack is as follows:

1) Find where the executable is loaded. Either 0x400000 for non-PIE executables (default) or stack read a saved return address.
2) Find a stop gadget. This is typically a blocking system call (like `sleep` or `read`) in the PLT. The attacker finds the PLT in this step too.
3) Find the BROP gadget. The attacker can now control the first two arguments to calls.
4) Find `strcmp` in the PLT. The attacker can now control the first three arguments to calls.
5) Find `write` in the PLT. The attacker can now dump the entire binary to find more gadgets.
6) Build a shellcode and exploit the server.

The attack requires a single (partial) scan of the executable. The PLT is the first part of an executable and is also the first item the attacker needs to bootstrap (*i.e.,* finding a stop gadget). The BROP gadget is found in the .text segment, which lives just after the PLT, streamlining the attack. After the BROP gadget, the attack is very efficient because it's a matter of scanning the PLT (relatively few entries) for two functions. The attack complexity is therefore based on the density of BROP gadgets and how long it takes to find the PLT.

I. First principles attack

One may think that eliminating BROP gadgets from executables or making the PLT difficult to find will stop BROP attacks. This is not the case and we present a less efficient

version of the attack that relies neither on the BROP gadget nor the PLT. The attack finds all the gadgets listed in Section VIII-A, namely the register pops and `syscall`. The attack outline is:

1) Find all `pop x; ret` gadgets.
2) Find a `syscall` gadget.
3) Identify the pop gadgets previously found.

The attacker starts by finding a stop gadget and all `pop x; ret` instructions. The difficulty is now in identifying the pop instructions and finding a `syscall` gadget. The idea is to identify the pop instructions based on system call behavior after tweaking system call arguments, in a similar way as to how `strcmp` was found in the optimized attack. There is a bootstrap problem, however, because to find `syscall` one must control the system call number (`rax`), so one must have a priori identified `pop rax; ret`.

The solution is to chain all pop instructions found by the attacker, popping the desired system call number, and one of them will likely be `rax`. The system call to use is `pause()` which takes no arguments and so ignores all other registers. It also stops program execution until a signal is raised and so it acts as a stop gadget, making it identifiable. The attacker can now append the probe address for `syscall` to the pop chain to find a system call gadget. Once an address that makes the program pause is found, the attacker can eliminate the pops one by one to find which one controls `rax`.

At this point the attacker has the address of a `syscall` gadget and a `pop rax; ret` gadget. The attacker also holds a list of unidentified pops. These are identified by using the following system calls:

1) First argument (`pop rdi`): `nanosleep(len, rem)`. This will cause a sleep of `len` nanoseconds (no crash). `rem` is populated if the sleep is interrupted, and it can be an invalid address as it is checked only after the sleep.
2) Second argument (`pop rsi`): `kill(pid, sig)`. If *sig* is zero, no signal is sent, otherwise one is sent (causing a crash). The *pid* need not be known: it can be zero which sends the signal to all the processes in the process group. To verify whether the signal is sent, the attacker can open multiple connections (going to different worker processes) to see if those connections are killed or not.
3) Third argument (`pop rdx`): `clock_nanosleep(clock, flags, len, rem)`. Similar to `nanosleep` but takes two additional arguments, making the third argument control the sleep length.

One can now call `write` and continue the attack by dumping the .text segment and finding more gadgets. While this attack is more general, it is more complex to perform because it requires two scans of the .text segment: one to find a list of pop gadgets, and one to find a syscall gadget.

A significant optimization is that all `pop rax; ret` gadgets we found were misaligned parses of `add rsp, 0x58; ret`. This information can be used to classify `pop rax` gadgets independently of `syscall` gadgets and significantly speed up the attack—one no longer needs to scan the entire .text segment twice. One can scan for the `add rsp, 0x58` gadget by setting up the stack with 11 traps followed by the stop gadget. To verify the gadget, the attacker jumps to the misaligned parse that yields `pop rax`, verifying that only one word is popped, which can be done by setting up the stack with a single trap followed by the stop gadget.

J. Other low-level details

In this section we list a number of not so obvious low-level attack details, many of which added to the attack's stability.

a) Stack reading with zeros: We found that an effective way to stack read is placing zeros in words like the saved frame pointer. It is likely to find an instruction pointer that does not crash the program regardless of the frame pointer. It also makes stack reading more robust when different worker processes are being used, each with a slightly different frame pointer. It may be impossible to finish reading a partially read frame pointer when being sent to a different worker process since all values will cause a crash. Forcing a zero word in this case will eliminate this problem.

b) Further `strcmp` verification: To further verify `strcmp`, we run it against the last byte of the vsyscall page, which is mapped at a static location. `strcmp` will terminate prior to reaching the end of vsyscall, not causing a crash. Most other functions instead will attempt to read past the vsyscall page causing a crash. This will prune functions that do not normally crash when supplied two readable arguments.

c) Dealing with small buffers: Sometimes attackers must minimize the length of ROP chains and be able to exploit small buffers. This situation occurs, for example, due to short reads or having to keep some memory intact (*e.g.,* not touching a canary), which limits the length of the overflow and the buffer space available. The yaSSL+MySQL exploit requires this optimization in order to avoid corrupting a canary. This is a checklist for conducting BROP with short ROP chains of at most 8 words (64 bytes):

- Find actual PLT entries based on their address, not based on their push number and slow path. This will make PLT invocation a shorter ROP chain.

- Dump the binary with a minimal ROP chain: strcmp address to dump, do not set `rsi` again (already set for strcmp), and call write. If zero is read, the dumped address contained a zero. Otherwise a small amount of the binary (up to a zero) will be read. Continue this until a `pop rdx` is found. After that use `pop rdx` to control the length rather than strcmp (shorter ROP chain).

- Create the shellcode environment in multiple stages: one connection to `dup` the attacker's socket, one to `read` "/bin/sh" into memory, and one to `execve`. All these connections (apart from `execve`) must terminate the ROP chain with a stop gadget to prevent a crash since the worker process is being prepared incrementally.

d) Dealing with few event-based workers: There are situations where an application is configured with very few event-based workers which can all become unresponsive during the

BROP attack, as they are all running the stop gadget (*e.g.,* an infinite loop) making it impossible to continue the attack. nginx is such an example and is configured with four workers by default. This is not a problem with applications that fork per connection or multi-threaded workers. In the latter case, one can connect and be handled by a new thread, and then send an exploit that causes a crash to kill any stuck threads in that process. Our general solution however is to try and use a stop gadget that returns to a higher call frame (*stack-based stop gadget*) rather than an infinite-loop-based one when possible. Our implementation uses the following algorithm, which works when at least three worker processes exist:

1) Find a PLT-based stop gadget (*e.g.,* sleep).
2) Find a PLT entry that returns. This is any PLT function that does not crash and does not cause an infinite loop (*i.e.,* does not act as a stop gadget). This function is useful because it mimics a *ret* instruction, allowing us to pop individual words from the stack (effectively acting as a `nop` in ROP). We will call this the "return gadget".
3) The first two steps of this algorithm will cause two workers to hang. Further hanging can now be avoided and the goal is to find a stack-based stop gadget by incrementally populating the stack with return gadgets. That is: attempt an exploit with a single return gadget. Then repeat with two, then three, and so on. Eventually, if a stack-based stop gadget is present, the application will not crash. The algorithm should give up if such gadget is not found after a certain depth, and if so, the PLT-based stop gadget must be used. If the gadget is found, however, the stack setup for the continuation of the attack will now be the ROP chain being probed, followed by a number of return gadgets to pad up to the depth found, which will then be followed by the stack-based stop gadget (not overwritten by the attacker).

The most conservative implementation that will work with even a single worker processes, stack reads higher stack frames to leak more return addresses, and attempts to return to those to see if program resumption occurs. This is rather inefficient, however, as a lot of stack may have to be read.

IX. IMPLEMENTATION

We implemented the BROP attack in a tool called "Braille" that automatically goes from a crash to a remote shell. It is written in 2,000 lines of Ruby code. Braille is essentially a meta-exploit that takes a driver function that can crash a server, and figures out all the information needed to build an exploit. The interface is as simple as:

```
try_exp(data) -> CRASH, NO_CRASH, INF
```

The driver needs to implement the `try_exp()` function and guarantee that when given enough "data" bytes it will end up overwriting a saved return address on the stack. The function returns CRASH if the socket closes after sending the data, NO_CRASH if the application appears to behave normally, or INF if the socket stays open for longer than a timeout. The timeout is automatically determined by the framework based on how quickly a crash is detected. The NO_CRASH

```ruby
def try_exp(data)
  s    = TCPSocket.new($victim, 80)

  req = "GET / HTTP/1.1\r\n"
  req << "Host: site.com\r\n"
  req << "Transfer-Encoding: Chunked\r\n"
  req << "Connection: Keep-Alive\r\n"
  req << "\r\n"
  req << "#{0xdeadbeefdead.to_s(16)}\r\n"
  req << "#{data}"

  s.write(req)

  if s.read() == nil
    return RC_CRASH
  else
    return RC_NOCRASH
  end
end
```

Figure 12. nginx exploit driver code. It merely wraps the input string provided by Braille into an HTTP chunked request and returns a CRASH signal if the connection closes.

return code is useful for probing the length of the buffer being overflowed or when stack reading. This return code is expected when no overflow occurs or when the same value was overwritten on the stack. The INF return code is expected when looking for gadgets as it indicates that the stop gadget ran ("infinite loop"). In later phases of the attack where data is expected from the socket, for example after `write` has been found and the binary is being dumped, a raw flag can be passed to return the actual socket instead of the CRASH / INF result code.

The driver code can be as simple as opening a socket to the server, and sending the data over the socket, raw. Often however, services expect data to be in a certain format. For example, HTTP headers may need to be present. The driver is responsible for this formatting. We wrote three drivers, all of which were under 100 lines of code. One passed data raw, one constructed an SSL packet and another one a chunked HTTP request. Figure 12 shows the code of a basic version of the nginx driver.

We also implemented a generic IP fragmentation router which is useful in cases where a single large TCP `read` is needed to overflow a buffer. For example overflowing a 4096-byte buffer with a single non-blocking `read` may be impossible with an MTU of 1500, as there may not be enough data queued, making the exploit unreliable. Our router instead sends large TCP segments as multiple IP fragments so that `read` is guaranteed to return a large packet, triggering the overflow reliably. We implemented it in 300 lines of C code. It creates a virtual `tun` interface where no TCP segmentation occurs—a single `write` is sent as multiple IP fragments and as a single TCP packet. This router was needed for nginx, for example.

Sending TCP segments out of order would be an alternative approach which may work and be more robust to firewalls.

```
ClientHello ch;

for (uint16 i = 0; i < ch.suite_len_; i += 3) {
  input.read(&ch.cipher_suites_[j], SUITE_LEN);
  j += SUITE_LEN;
}

input.read(ch.session_id_, ch.id_len_);

if (randomLen < RAN_LEN)
  memset(ch.random_, 0, RAN_LEN - randomLen);

input.read(&ch.random_[RAN_LEN - randomLen],
           randomLen);
```

Figure 13. Vulnerable code in yaSSL. The attacker controls `suite_len_`, `id_len_` and `randomLen`. The `ClientHello` buffers are fixed sized, on the stack. Using `randomLen` as an attack vector lets an attacker overwrite, via pointer arithmetic, a return address without having to touch the canary. `randomLen` cannot be too large or it will hit the canary, and so only a small buffer is available for the exploit in this condition.

X. EVALUATION

We tested the BROP attack in three scenarios:

1) An open-source SSL library with a known stack vulnerability (yaSSL). This mimics the scenario where one is attacking a proprietary service that is believed to use a vulnerable open-source component. As a sample target we used an older version of MySQL that used yaSSL.

2) An open-source software with a known stack vulnerability (nginx), manually compiled from source. In this scenario the attacker knows the source of the entire server but does not hold the binary.

3) A toy closed-binary proprietary service with a stack vulnerability. This was written by a colleague and both the binary and source were kept secret. Ideally we would have tested this against a real-world proprietary service but it would have been difficult to do so legally.

The vulnerabilities were as follows.

a) yaSSL: Figure 13 shows the vulnerable code in yaSSL. The SSL hello packet has variable length values for the cipher-suite, session id, and client random. The lengths are specified in the packet, and yaSSL copies these values into fixed sized buffers on the stack. This makes it possible to exploit the program in three ways by overflowing any of the buffers. Interestingly, the copy for the client random contains pointer arithmetic which makes it possible to write on the stack starting from the saved return address instead of being forced to overwrite the canary. This is important for MySQL where the server is re-executed after a crash and so stack reading for canaries would not work because the canary will have changed. The caveat of using this approach is that the size of the overflow is small in practice: if a large buffer is used (large `randomLen`) then the canary will be overwritten. One is therefore forced to use short buffers. Braille is implemented carefully to support short buffers by using short ROP chains.

b) nginx: HTTP requests can be sent as chunks, where each chunk has a length, followed by the chunk data. If a large chunk value is supplied, it will be stored in a signed variable

```
typedef struct {
  ...
  off_t content_length_n;
} ngx_http_headers_in_t;

u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];

size = (size_t) ngx_min(
            r->headers_in.content_length_n,
            NGX_HTTP_DISCARD_BUFFER_SIZE);

n = r->connection->recv(r->connection, buffer, size);
```

Figure 14. Vulnerable code in nginx. The attacker controls `content_length_n` (signed) and can supply a negative value. `size` will be unsigned, resulting in a large number if `content_length_n` is negative. `buffer` is 4096 bytes long, on the stack.

TABLE II. CUMULATIVE NUMBER OF REQUESTS PER BROP ATTACK PHASE.

Attack phase	Proprietary server	yaSSL + MySQL	nginx
Stack reading	1028	406	846
find PLT	1394	1454	1548
find BROP gadget	1565	1481	2017
find `strcmp`	1614	1545	2079
find `write`	1624	1602	2179
dump bin & exploit	1950	3851	2401
Time (min)	5	20	1

resulting in a negative number. A check is performed if the size (now negative) is smaller than the buffer size. The size is then cast to an unsigned value passed as the length parameter to `read`, making it possible to read a large chunk into a 4096-byte buffer on the stack. Figure 14 shows the vulnerable code in nginx.

c) Proprietary service: When sending a short string, "OK" was read from the server. On a very long string, the connection closed.

We ran Braille against all three attack scenarios, without any application-specific optimizations, and the attack succeeded in all cases. We evaluate the following aspects:

1) Performance: number of requests and time.
2) Stability: how robust the attack is.
3) Attack paired with source-code knowledge: whether having access to the source code (but not the binary) can make the attack better.

A. Performance

Table II shows the cumulative number of requests needed for each attack phase. The attack can complete in under 4,000 requests, or 20 minutes. This is acceptable given that in the past attacks on 32-bit ASLR required on average 32,768 requests [9]. In all cases, in about 1,500 requests the attack is able to defeat canaries, ASLR and find a stop gadget. That's how long it takes to go from zero knowledge to being able to execute a useful code fragment. From then on, it's a matter of finding the BROP gadget which, depending on its popularity, can take between 27 to 467 requests.

Table III shows how popular the BROP gadget is and how many probes are expected to find it in a sample set of binaries.

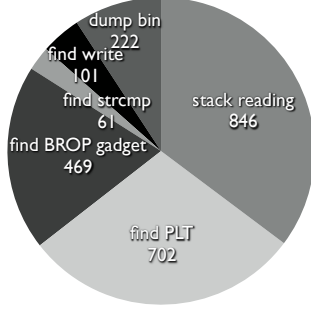| Binary	BROP count	expected scan length (density)
proprietary service	194	154
MySQL	639	501
nginx	130	566
Apache	65	860
OpenSSH	78	972

Figure 15.   Attack complexity for nginx. The number of requests needed for each phase are shown. Broadly speaking, the attack's complexity is split in four parts: stack reading, finding the PLT, finding the BROP gadget, and dumping the binary to finish the attack.

The data shows the number of BROP gadgets present, and their density: $\frac{.text\_size}{7 \times BROPcount}$ (recall that 7 bytes can be skipped per probe due to the size of the gadget). The BROP gadgets appears very popular and can be found in under 1,000 address probes. Note that in practice more requests will needed to verify the gadget and weed out false positives.

After the BROP gadget is found, finding `write` takes only a few additional requests, and can usually be found in approximately 2,000 requests total. At this point the attack is almost complete. One may choose to manually `write` very specific parts of the binary to minimize the number of requests based on the information learned. Otherwise, our Braille tool starts dumping the binary from its start, until the entire symbol table is dumped so that a shellcode can be constructed. The attack typically completes within 500 additional requests (about 2,500 total). In the case of yaSSL, it took many more requests to dump the binary because the buffer being overflowed was very short and so Braille was limited in how long the ROP chain could be. Braille was forced to dump the binary in small chunks to find a `pop rdx; ret` (a rare gadget) before the rest of the binary could be downloaded in larger chunks.

Figure 15 shows the complexity of the attack for nginx. The attack's overhead can be split into four parts: stack reading (35%), finding the PLT (29%), finding the BROP gadget (20%) and finishing off (16%). Note that if canaries are not used (or can be bypassed, like in yaSSL) and the PIE flag is not used (the default) then stack reading can be avoided altogether. Finding the PLT largely depends on the size of the executable and how many PLT entries are skipped during a scan. The BROP gadget scan will depend on its frequency, as previously mentioned.

The attack can complete within 20 minutes. MySQL took a long time because it took a while for it to restart after each crash. nginx was fastest (only one minute) because a non-time based stop gadget was used. An HTTP keep-alive connection was used and so after the exploit request, a normal request was sent to check if the connection was still alive. In the proprietary server case instead, a timeout had to be used to determine if the server was still alive which made the attack slower.

The attack clearly is noisy but we argue that if it executes fast enough, the attacker may be able to perform whatever activity he needs to do before getting caught. nginx for example logs each crash, in a file owned by root. The server runs as nobody so the attacker would not be able to erase the logs. We notice, however, that the worker processes keep file descriptors to the logs open, making it possible to write a shellcode to call `ftruncate` to erase traces of the attack.

B. Stability

The three servers use worker processes very differently, exercising BROP in different ways. In all cases the BROP attack was reliable and completed unassisted without hanging or causing denial-of-service.

MySQL is (typically) single process, multi-threaded. On a crash, a script (mysqld_safe) reexecutes the server. The BROP attack works under the default configuration (no PIE, but canaries) despite the re-execution because the canary is never hit thanks to how the bug is being exercised. If compiled with the PIE flag, the attack would not work as one couldn't read a (changing) return address from the stack to defeat ASLR. This does not apply to nginx and the toy proprietary service where due to their `fork`ing nature, the attack would succeed even when PIE is used.

nginx has multiple worker processes and has a single-threaded, event-based architecture. Most distributions configure four worker processes by default. This makes it a tricky scenario because an infinite loop based stop gadget would hog the worker completely, and one gets only four shots by default. The stop gadget here was returning to a higher stack frame, which avoided any blocking. With a specialized exploit, we are able to exploit nginx even when configured to use a single worker.

The proprietary server `fork`ed once per connection. This makes the attack very reliable as there is a virtually infinite number of worker processes available. We did not know *a priori* about the details of the server but it contained a few unique things. The stack overflow was one stack frame above the actual bug as there was a function that wrapped the read system call. The server also contained a single loop, dependent on a variable used to exit the loop when shutting the service down. This created the additional challenge that the loop was not easily usable as an infinite loop gadget.

The stop gadgets for yaSSL+MySQL, nginx and the proprietary server respectively were: `futex`, returning to a higher call frame, and `sleep`.

The yaSSL+MySQL scenario offered a very small overflow buffer and shows that BROP can work even with small buffers (64 bytes are sufficient).

The key to the success and stability of the BROP attack is that the attacker needs to scan for a single item at any given

time. There are no dependencies between the items required for the attack. Also, the items that are being looked for have very prominent signatures (*e.g.,* the PLT and BROP gadget) and no false positives were found during the attack. The attack also gets more robust as it progresses.

C. Source-code aid

If the attacker has some knowledge of the server (*e.g.,* source code) the number of requests may be brought down. In nginx for example, the attacker does not need to find a stop gadget because one is already present in a higher stack frame. The attacker also has an idea of how large the binary is and how many PLT entries there should be, making the PLT scanning faster, *e.g.,* by skipping more entries and starting closer to the PLT. In the case of nginx, the overflow happens right after a read call so rdi (the first argument) has the socket number. The attacker can exploit this fact by calling `write` without setting rdi or calling `dup2` to copy the file descriptor to a fixed number; by comparison the generic attack must brute-force the file descriptor number. Knowing that reads are non-blocking is also useful so that the IP fragmentation router is used to trigger the overflow. With all this in mind, we wrote an nginx optimized version of the BROP attack that took under 1,000 requests on a default Debian install (no canary or PIE). This exploit should work on any distribution as it is not binary-specific.

Source knowledge helped in the yaSSL case too. The bug could be exploited in three ways, but only one allowed circumventing canaries. All three vulnerabilities would have to be revealed through remote fuzz testing and triggered independently. BROP would only then succeed on the single vulnerability where stack reading succeeds: the one where canaries are not touched.

Based on source information, an attacker may determine whether rdx has a sane value at the time of exploit. If so, the attacker may skip having to find `strcmp` to control rdx and proceed with a more optimal attack.

We made some discoveries while blindly hacking the toy proprietary service that would have been apparent upfront given the source code and simplified the attack. During the stack reading phase, we noticed that the number 4 was present on the stack. This indeed was the socket number and the attack could have avoided a file descriptor brute-force later on. Also, when attempting to read the saved return address, a particular return address was found that would force "OK" to be written to the network. This could have been used as a stop gadget avoiding having to scan for one.

Stack reading is a very useful tool when hacking blind. It reveals whether canaries are used, frame pointers are enabled, and possibly more. This helps in fingerprinting the distribution (based on defaults) or the environment being hacked. Stack reading in the yaSSL case also noted revealed that the overflow was occurring in the "opposite" direction due to pointer arithmetic—the first byte of the data being sent, rather than the last one, was affecting whether the program crashed or not.

XI. LIMITATIONS

The BROP attack has its limitations. We applied it only to simple stack overflows. While it is a good starting point, many vulnerabilities are more complex and heap-based.

Stack reading assumes that the attacker can overflow at a byte granularity and controls the last byte being overflown (*e.g.,* a zero is not appended by the server).

The attack assumes that the same machine and process can be hit after each attempt. Load balancers can cause the attack to fail when PIE is used and canaries cannot be circumvented.

The attack also relies on a number of workers being available and not ending up in a situation where all workers become "stuck" in an infinite loop. This makes the stop gadget selection very important. Returning to a higher stack frame is a key optimization here, where the worker is "resumed" rather than caused to hang. If this cannot be done and there are a limited number of worker processes, and the stop gadget hangs them indefinitely, the attack may not complete. nginx is an example where this can happen as it can be configured with a single worker and is event-based. However, BROP still succeeds here because it is possible to return to a higher stack frame.

XII. DISCUSSION

A. BROP in different OSes

Windows lacks a `fork`-like API (it has only `CreateProcess`) so canaries and the text segment's base address are guaranteed to be rerandomized after a crash, making the system more robust against BROP-like attacks. The Windows ABI also passes arguments in scratch registers (*e.g.,* rcx, rdx) making pop gadgets for them harder to find. Gadgets involving scratch registers are rare because they are not preserved across function calls, so the compiler does not need to save them to the stack. Such gadgets will likely only exist as misaligned parses, making them less likely.

ASLR implementations vary by OS. Windows 8.1 and Mac OS X randomize everything by default. Unfortunately, both systems rerandomize system libraries only at reboot time. This can create a BROP-like environment for leaking pointers to system libraries. Reboots can be rare, especially on clients and laptop systems where users may suspend and resume more often than reboot. Mac OS X also only supports 16-bits of entropy for ASLR, placing it far behind other 64-bit operating systems. On Linux, the effectiveness of ASLR depends on the distribution and its PIE configuration. For example, Ubuntu does not enable PIE by default, but has enabled it on a per-application basis based on risk [16].

B. BROP beyond stack overflows

The BROP attack focuses on stack attacks, the simplest possible scenario. We did not consider heap exploits, though these might be possible. The bootstrap, for example, would be different, as a stack pivot gadget would have to be found after the stop gadget. A useful stack pivot gadget would be, for example, `mov rax, rsp; ret` assuming that `rax` points to an attacker-controlled buffer at the time of exploit. The attacker can now set up a stack and ROP chains in that buffer.

TABLE IV.   CODE DIVERSITY WHEN THE SAME VERSION OF NGINX (1.4.0) IS COMPILED WITH DIFFERENT DEBIAN LINUX VERSIONS.

	Text Size	Text Start	# of Gadgets
Squeeze	0x5fc58	0x4031e0	206
Wheezy	0x61f0c	0x4032f0	255
Jessie (testing)	0x5fbd2	0x402ee0	323

C. Client-side vs. server-side

It may be possible to launch a BROP-like attack on clients. Browsers like Chrome, for example, launch plugins in a separate process for robustness. JavaScript can be used to create multiple vulnerable plugin objects, attempt an exploit, and detect whether they have crashed or not without user interaction. We note, however, that there is typically lower hanging fruit on the client-side. Having the execution power of JavaScript available can offer more signaling mechanisms to the attacker compared to a coarse-grained crash / no-crash as used in server-side BROP.

An interesting distinction between client-side and server-side is that often client-side attacks are less targeted. For example, an attacker may want to own any given number of clients to steal information or construct a botnet. This makes exploits for older targets with fewer protections (*e.g.,* Windows XP) still valuable, as there still are people running those configurations. Server-side attacks instead are often targeted as one wants to attack a particular site. Relying on 32-bit targets or specific binary installations, or simply moving on to the next victim may not be an option. This makes BROP very valuable on the server-side as it gives an attacker a larger hammer when needed.

D. Variance in binaries

Counterintuitively, closed-source systems (though open-binary) make writing exploits simpler. Many exploits that target Windows are very robust as they build ROP chains on DLLs that seldom change, and so only a few versions exist. In an open-source setting, there are multiple binary variants and the attacker must build a different ROP chain for each. Table IV shows the size and start address of different distributions of the exact same nginx version. As we see there is a lot of variability based on the build environment, the version of the libraries it was linked against, and the compiler version, even though the same Linux distribution was being used. Even a single byte difference or offset will defeat a statically precomputed ROP chain.

Worse for the attacker, a system may be manually compiled by the end user, making it impossible for the attacker to build a ROP chain offline as the binary is unknown. In such cases BROP is a necessity. Even if a server uses a precompiled binary, it can be difficult to determine which particular one is being used: remote OS fingerprinting reveals an approximate kernel version, not a distribution. BROP in fact can be used to fingerprint distributions and applications (*e.g.,* based on whether canaries are present, vsyscall behavior, *etc.*).

E. Remote fuzz testing

The BROP attack could be a powerful tool for hacking proprietary closed-binary services when coupled with a remote fuzz tester. We note that in two of the example applications we targeted, the overflow occurred because a length was specified in the packet but a larger value was sent. It certainly is possible to write a fuzz tester that has knowledge about a protocol and attempts to overflow by supplying incorrect lengths [17]. Interestingly, pretty much the same chunked encoding vulnerability that appeared in nginx has already appeared in Apache in the past [18]. It may be possible to write fuzz testers for particular protocol conditions that are known to be hard to implement correctly, or that have been known to be exploited in the past.

XIII.   BROP PREVENTION

The following is a discussion of defense mechanisms that will prevent the BROP attack, including two precautions we suggest server developers use. There is a lot of prior research in ROP attack defense mechanisms, and many of those techniques are applicable to defending against BROP. Thus, this list is by no means comprehensive.

A. Rerandomization

The most basic protection against the BROP attack is to rerandomize canaries and ASLR as often as possible. These protection mechanisms are effective, but server developers undermine them by not rerandomizing when worker processes crash. The simplest method is to fork and exec the process on a crash or spawn, which rerandomizes both the canary and ASLR. It is important that any child processes forked are randomized independently so that any information learned from one child cannot be used against another one.

There has been research on rerandomizing binaries at runtime. One such technique is work by Giuffrida et al. that uses a modified compiler to migrate the running state between two instances (with a different ASLR randomization) [19]. We also prototyped a re-randomization technique that moves a binary's text segment to a new location using mmap/munmap, and uses a page fault handler to determine whether pointers should be rewritten as they are faulted on.

An even simpler improvement we developed is to rerandomize the canary on a per-user or per-request basis. We suggest servers write a new canary before entering a per-request function. On the return through that function the old canary should be restored so that execution can continue normally. While this protects against the bugs in nginx and our proprietary server, the particular attack against yaSSL can avoid the canary entirely.

B. Sleep on crash

Systems like NetBSD's segvguard [20] and grsec's `deter_bruteforce` for Linux [21] propose delaying a `fork` after a segmentation fault. This technique can slow down attacks such that an administrator can notice something is wrong and address the problem. The downside of this approach is that bugs now can become easy denial of service attacks. It is also unclear what a good value for the delay is. grsec proposes a 30 second delay. While this is sufficient for most setups, overnight attacks on a small site might go unnoticed: our optimized BROP attack for nginx can complete in 1,000 requests, making the attack time roughly 8 hours.

While denial of service attacks are serious, leaking private data can be even worse. In some situations, servers should not respawn, but in practice users and developers find this an unacceptable solution. Modern Linux desktops use *systemd* to monitor services and automatically restart services on failures. Developers should be cautious about which remote services they really need or want to restart automatically to reduce the attack surface.

C. ROP protections

Another class of defense mechanisms is protections that defend against ROP attacks. Firstly, Control Flow Integrity (CFI) [22] prevents return oriented programming in general by enforcing the control flow graph. There are many other similar techniques.

Another approach developed by Pappas et al. is to enforce control flow inside the system call handler by comparing the stack against last branch record (LBR) facility available in Intel processors [23]. This can be used to verify that the stack has not been tampered with. The main limitation is the depth of the stack that can be checked is as small as four entries depending on the processor model.

There are solutions that propose adding randomness to binaries [24]. While these are effective against ROP they are not in a BROP setting. Additional techniques exist to try to randomize gadget locations on a per instance run, but these offer no defense against BROP unless the binary is fully restarted (fork and exec) [25], [26]. There are also techniques to remove or reduce the number of available gadgets [27] that could protect effectively against ROP attacks in general.

D. Compiler Techniques

Many modern compilers support inserting runtime bounds checks on buffers. This prevents a class of bugs, unlike canaries that detect corruption after the fact or not at all if a successfully hacker brute-forces the secret value. LLVM contains *AddressSanitizer*, a tool for bounds checks, and use after free bugs [28]. The SafeCode framework is built on LLVM and also enforces bounds checks among other things [29]. Intel compilers also provide support for runtime bounds checking.

The main problem with all these solutions is that they may suffer as much as a 2x performance slowdown, and as such they are used mostly for testing. One bright spot to make these solutions practical is that Intel has announced a set of instruction extensions to reduce the cost of bounds checking variables [30].

XIV. Related work

Prior work exists on scanning for a single ROP gadget. Goodspeed's half-blind attack against microcontrollers [31] relies on knowledge of (common) bootloader code, and scanning for a single gadget in an unknown portion of memory to construct an attack used for firmware extraction. The BROP attack is more generic as it is fully blind and presents techniques to find and chain multiple, different, gadgets.

There has been work on attacking instruction set randomization [32] that uses similar techniques to the BROP attack. In that work, signaling for correct/incorrect guesses was based on whether an application crashes or not. The goal, however, was to leak an encryption key, and the method assumes that a code injection exploit can be carried out in the first place: *i.e.,* no ASLR and no NX were in place.

Stack reading to determine canaries is a well known technique [33]. Researchers as well as attackers have shown how to brute-force 32-bit ASLR [9], but this approach was not feasible on 64-bit machines as it brute-forced the entire word at once. We generalize the stack reading technique to reading off the saved return address and frame pointer to break 64-bit ASLR.

Many have noted that exploits today are multistage and require leaking information. Kingcope's 32-bit nginx exploit for example brute-forces the location of write in the PLT to leak the binary to find gadgets and construct a binary-agnostic exploit [14], [34]. This technique falls short on 64-bit because multiple items need to be brute-forced at once: the location of write and all the gadgets needed to populate the arguments to write (the latter was not needed on 32-bit). This makes a BROP-like attack a necessity on 64-bit; even Kingcope admits difficulty in generalizing his approach to these platforms. The author also admits that the exploit does not work on WANs due to nginx's non-blocking read. The missing link was IP fragmentation. The nginx exploit is a great case study showing the problems involved when writing modern server-side exploits end-to-end.

Client-side exploit writers have had more luck with 64-bit and ASLR, at least publicly, possibly due to contests that highly reward participants and force them to publish their work [35]. This year's pwn2own exploit uses a JavaScript vulnerability to leak a pointer, and then uses the same vulnerability to leak the entire contents of the chrome.dll library to build a ROP chain. Again, this shows how exploits are moving toward being binary-independent for robustness. This made a difference even in closed-source systems where there are relatively fewer versions of the binary, because Chrome released a new version of the DLL shortly prior to the contest demo, which would have stopped any exploit based on a specific binary.

XV. Conclusion

We show that, under the right conditions, it is possible to write exploits without any knowledge of the target binary or source code. This works for stack vulnerabilities where the server process restarts after a crash. Our attack is able to defeat ASLR, NX and stack canaries on modern 64-bit Linux servers. We present two new techniques: generalized stack reading, which defeats full ASLR on 64-bit systems, and the BROP attack, which is able to remotely find ROP gadgets. Our fully automated tool, Braille, can take under 4,000 requests to spawn a shell, under 20 minutes, tested against real versions of yaSSL+MySQL and nginx with known vulnerabilities, and a toy proprietary service running an unknown binary.

We show that design patterns like forking servers with multiple worker processes can be at odds with ASLR, and that ASLR is only effective when it is applied to all code segments in the binary (including PIE). Moreover, security through obscurity, where the binary is unknown or randomized, can only slow but not prevent buffer overflow attacks. In order

to defend against our attack, we suggest that systems should rerandomize ASLR and canaries after any crash, and that no library or executable should be exempt from ASLR.

Braille is available at: http://www.scs.stanford.edu/brop/.

ACKNOWLEDGMENTS

REFERENCES

[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133375.2133377

[2] mitre. Cve-2013-2028. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028

[3] ——. Cve-2008-0226. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0226

[4] A. One, "Smashing The Stack For Fun And Profit," *Phrack*, vol. 7, no. 49, Nov. 1996. [Online]. Available: http://phrack.com/issues.html?issue=49&id=14#article

[5] M. Kaempf. Vudo malloc tricks by maxx. [Online]. Available: http://www.phrack.org/issues.html?issue=57&id=8&mode=txt

[6] S. Designer. Getting around non-executable stack (and fix). [Online]. Available: http://seclists.org/bugtraq/1997/Aug/63

[7] P. Team. Pax address space layout randomization (aslr). [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[8] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251361

[9] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[10] gera and riq. Advances in format string exploitation. [Online]. Available: http://www.phrack.org/archives/59/p59_0x07_Advances%20in%20format%20string%20exploitation_by_riq%20&%20gera.txt

[11] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267554

[12] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks (ProPolice)," 2003, http://www.trl.ibm.com/projects/security/ssp/. [Online]. Available: http://www.trl.ibm.com/projects/security/ssp/

[13] Bulba and Kil3r, "Bypassing stackguard and stackshield," *Phrack Magazine*, May 2000. [Online]. Available: http://phrack.org/issues.html?issue=56&id=5#article

[14] Kingcope. About a generic way to exploit linux targets. [Online]. Available: http://www.exploit-db.com/wp-content/themes/exploit/docs/27074.pdf

[15] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 60–69. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2009.16

[16] Ubuntu security features. [Online]. Available: https://wiki.ubuntu.com/Security/Features

[17] Peach fuzzer. [Online]. Available: http://peachfuzzer.com/

[18] mitre. Cve-2002-0392. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0392

[19] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 40–40. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362833

[20] E. Efrat. Segvguard. [Online]. Available: http://www.netbsd.org/~elad/recent/man/security.8.html

[21] grsecurity. Deter exploit bruteforcing. [Online]. Available: http://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options#Deter_exploit_bruteforcing

[22] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[23] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 447–462. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534805

[24] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382216

[25] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 571–585. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.39

[26] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 601–615. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.41

[27] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.

[28] T. C. Team. Addresssanitizer - clang 3.4 documentation. [Online]. Available: http://clang.llvm.org/docs/AddressSanitizer.html

[29] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 144–157. [Online]. Available: http://doi.acm.org/10.1145/1133981.1133999

[30] Intel. Introduction to intel memory protection extensions. [Online]. Available: http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[31] T. Goodspeed and A. Francillon, "Half-Blind Attacks: Mask ROM Bootloaders are Dangerous," in *WOOT*, 2009.

[32] A. N. Sovarel, D. Evans, and N. Paul, "Where's the feeb?: The effectiveness of instruction set randomization," in *Usenix Security*, 2005.

[33] A. Zabrocki. Scraps of notes on remote stack overflow exploitation. [Online]. Available: http://www.phrack.org/issues.html?issue=67&id=13#article

[34] Kingcope. nginx 1.3.9/1.4.0 x86 brute force remote exploit. [Online]. Available: http://www.exploit-db.com/exploits/26737/

[35] M. Labes. Mwr labs pwn2own 2013 write-up - webkit exploit. [Online]. Available: https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/