A REVIEW PAPER ON HACKING BLIND

Chanchal Chawla*

* Teerthanker Mahaveer Institute of Management and Technology,
Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, INDIA
Email id: chanchal.management@tmu.ac.in

DOI: 10.5958/2249-7137.2021.02484.8

ABSTRACT

The author demonstrate how to build remote stack buffer overflow attacks against services that resume after a crash without having a copy of the target binaries or source code. This enables the hacking of proprietary closed-binary services as well as open-source servers that have been manually built and installed from source and whose binary is unknown to the attacker. Traditional methods are typically used in conjunction with a certain binary and distribution where the hacker is aware of the location of relevant Return Oriented Programming gadgets (ROP). Instead, our Blind ROP (BROP) attack identifies enough ROP gadgets across the network to execute a write system call and transmit the vulnerable binary, after which an exploit may be carried out using existing methods. This is done by leaking a single piece of data depending on whether or not a process failed when given a certain input string. Stack vulnerability and a service that restarts after a crash are required for BROP to work. We used Braille, a fully automated exploit that yielded a shell in under 4,000 requests (20 minutes) against a current nginx vulnerability, yaSSL + MySQL, and a toy proprietary server written by a colleague, against a contemporary nginx vulnerability, yaSSL + MySQL, and a toy proprietary server written by a colleague. The attack uses address space layout randomization (ASLR), no-execute page protection (NX), and stack canaries to operate against current 64-bit Linux.

KEYWORDS: *ASLR, Attack, Blind, Hacking, ROP.*

1. INTRODUCTION

Attackers have had a lot of success developing exploits with different degrees of target knowledge. Because attackers may audit the code to discover weaknesses, open-source software is the most vulnerable. Fuzz testing and reverse engineering may also be used to hack closed-source software for highly motivated attackers. In order to better comprehend an attacker's capabilities, we ask: can attackers expand their reach and develop vulnerabilities for proprietary services when neither the source nor binary code is available? This objective may seem unachievable at first glance, since today's exploits depend on obtaining a copy of the target binary to utilize in Return Oriented Programming (ROP).Because non-executable (NX) memory protection has essentially prohibited code injection attacks on contemporary systems, ROP is required **[1].**

To begin answering this issue, we'll look at the most basic vulnerability: stack buffer overflows. Unfortunately, these vulnerabilities are still present in popular software today (for example, nginx CVE-2013-2028). It's just a guess that vulnerabilities like this go undetected in private software, since the source (and binaries) hasn't been subjected to rigorous examination by the general public and security experts. However, an attacker may definitely employ fuzz testing to discover potential vulnerabilities in known or reverse engineered service interfaces. Attackers may even use known flaws in popular opensource libraries (such as SSL or a PNG parser) that are utilized by proprietary services. The difficulty is devising a technique for exploiting these flaws when knowledge about the target binary is restricted. One advantage attackers often have is that many servers, for the sake of resilience, restart their worker processes after a crash. Apache, nginx, Samba, and OpenSSH are just a few examples **[2].**

Even if this capability is not built into the program, wrapper scripts like mysqld safe.sh or daemons like system offer it. Load balancers are becoming more popular, and they often distribute connections to a large number of similarly configured servers running identical application binaries. As a result, there are numerous scenarios in which an attacker has an unlimited number of attempts (until discovered) to develop an exploit **[3].**

Blind Return Oriented Programming (BROP) is a novel technique that takes advantage of these circumstances to create vulnerabilities for proprietary services whose binaries and source are unknown. The BROP attack assumes a stack vulnerability in a server application that is restarted after a crash. The attack works with ASLR (Address Space Layout Randomization), nonexecutable (NX) memory, and stack canaries enabled on contemporary 64-bit Linux. While this covers a huge number of servers, we are unable to target Windows systems at this time due to the fact that we have not yet adapted the attack to the Windows ABI.

1.1 Two novel methods are used to enable the attack:

- Generalized stack reading: this extends a well-known method for leaking canaries to additionally leak stored return addresses, allowing ASLR to be defeated on 64-bit systems even when Position Independent Executables (PIE) are employed.
- Blind ROP: This method locates ROP devices from afar.

Both methods rely on a single stack vulnerability to leak data depending on whether or not a server process fails. The stack reading method successfully reads (by overwriting) the stack by overwriting it byte-by-byte with various guess values until the right one is discovered and the server does not crash. The Blind ROP attack locates enough gadgets remotely to execute the write system call, which allows the server's binary to be copied from memory to the attacker's socket. Canaries, ASLR, and NX have all been overcome at this stage, and the exploit may now be carried out using known methods.

➢ *Three additional situations are enabled by the BROP attack, which allows for strong, general-purpose exploits:*

- Exploiting closed-binary services that are proprietary. When utilizing a remote service, one may detect a crash or find one via remote fuzz testing.
- Exploiting a flaw in an open-source library that is believed to be utilized in a closed-binary service. For example, a popular SSL library may have a stack vulnerability, and it's possible that it's being utilized by a proprietary business.

- Attempting to hack an open-source server whose binary is unknown. This is true for source-based distributions like Gentoo, as well as manually built installs.

We assess each of the three possibilities. In an ideal world, we would test our methods against production services for which we have no knowledge of the program, however we are limited by legal constraints. To mimic such a situation, we tested against a toy private service written by a colleague for which we had no source, binaries, or functionality details. In the second case, we take use of a genuine flaw in the yaSSL library. This library was formerly used by MySQL, and we are using it as the host application **[4–7].**

In the third case, we build a generic attack for a recent (2013) vulnerability in nginx that is not dependent on a specific binary. This is especially helpful since the exploit works with every distribution and vulnerable nginx version, eliminating the need for an attacker to create a separate exploit for each distribution and version combination (as is done today).

We built Braille, a new security tool that greatly automates BROP attacks. Braille can get a shell on a vulnerable server in around 4,000 queries, which takes less than 20 minutes in most cases and just a few minutes in others. To crash the server, an attacker simply has to supply a function that creates a request of a minimum length and appends a string supplied by Braille. The function must additionally return a single bit indicating whether or not the server has crashed.

1.2 Our contributions are as follows:

- A server-side method to bypass ASLR (generalized stack reading).
- A method for remotely locating ROP devices (BROP) in order to attack software when the binary is unknown.
- Braille: a program that generates an attack based on information on how to cause a stack overflow on a server.
- The first published attack for nginx's current vulnerability that is generic, 64-bit, and overcomes (full/PIE) ASLR, canaries, and NX (to our knowledge).
- Recommendations for countering BROP assaults. In conclusion, ASLR must be applied to all executable segments (PIE), and re-randomization must be performed after each crash (at odds with fork-only servers). Keeping the binary from the attacker or changing it on purpose may not be a good security countermeasure.

1.3 An Introduction to Buffer Overflows:

Buffer overflows are a well-known issue that has been exploited in the past. They are quite simple to tackle conceptually. A susceptible application, for example, might read data from the network and store it in a buffer. An attacker may then overwrite memory beyond the end of the buffer if the application has adequate bounds checks to restrict the amount of the incoming data. As a consequence, essential control-flow state may be changed, such as return addresses or function pointers. Return addresses are automatically close in memory owing to function calling practices, making stack buffer overflows particularly hazardous. Attacks against the heap's buffers, on the other hand, are possible **[8].**

It was standard practice in the early days of stack buffer overflows for an attacker to embed malicious code in the payload used to overrun the buffer. As a consequence, the attacker may simply set the return address to a known stack position and execute the instructions in the buffer. On current computers, such "code injection" assaults are no longer feasible since modern processors and operating systems may now designate data memory pages as non-executable (e.g., NX on x86). As a consequence, attempting to execute code on the stack will only result in an exception.

To overcome non-executable memory protections, a new approach known as return-oriented programming (ROP) was created. It operates by connecting small code snippets already existing in the address space of the application.

Gadgets are bits of code that may be combined to create arbitrary computations. As a consequence, without relying on code injection, attackers may utilize ROP to take control of applications. Simpler ROP variants are sometimes available. Return-to-libc attacks, for example, may utilize a high-level library function as the return address. The system() method, in particular, is handy for attackers since it may execute arbitrary shell code with only one parameter.

On 32-bit systems, when arguments were passed on the stack and were already within the attacker's control, these attacks were extremely successful. Arguments are given in registers on 64-bit platforms, thus extra gadgets are required to fill registers.As an additional protection against buffer overflow attacks, address space layout randomization (ASLR) was added. It operates by relocating code and data memory segments in the process address space in a random order. Code segment randomization is often used exclusively on libraries, although complete address space randomization is also feasible **[9], [10].**

Because ASLR makes the address locations of code (or even the stack) difficult to anticipate in advance, it presents a significant barrier for attackers. Unfortunately, ASLR is limited on 32-bit systems by the amount of available bits for randomization (typically 16). As a consequence, brute-force assaults may be very successful. On 64-bit systems, however, there are usually too many random bits for brute-forcing to work. In such instances, ASLR may still be bypassed, but only when used in conjunction with a vulnerability that exposes address space layout information, such as a format string.64-bit systems offer a further difficulty for attackers, in addition to the wider address space for ASLR and the requirement to find extra gadgets to fill argument registers. Because the architecture only allows for 48-bit virtual addresses, user-level memory pointers must include zero-valued bytes. Overflows caused by string operations like strcpy are terminated early due to these zeros().

Another popular buffer overflow protection is the use of canaries. Canaries cannot prevent buffer overflows, but they can detect them and stop the program before an attacker has a chance to manipulate control flow. With stack canaries, for example, a secret value chosen ahead of time is put immediately before each stored frame reference and return address. The secret value is then verified to ensure it has not changed when a function returns. Because an attacker must properly replace the secret value in order for the application to actually utilize an altered return address, this may prevent stack buffer overflows from being exploited.

Canaries, like ASLR, may be bypassed by exploiting an extra vulnerability that exposes information about the secret value. For canary implementations, the arrangement of stack memory is an essential issue. One typical technique is to put all buffers near the top of the frame, such that if they overflow, other variables will not be overwritten until the canary is corrupted. The reason for this is to prevent pointers from being exploited to overwrite arbitrary memory. Unfortunately, even with layout measures, the structure of a buffer overflow may occasionally allow an attacker to circumvent canary words and get direct access to crucial state, as occurred with unsafe pointer arithmetic in yaSSL.

2. DISCUSSION

The author has discussed about the hacking blind, Because Windows doesn't have a fork-like API (just CreateProcess), canaries and the text segment's base address are guaranteed to be re-randomized after quite a crash, making the system more resistant to BROP-style assaults. Arguments are also sent in scratch registers (e.g., rcx, rdx) by the Windows ABI, makes pop gadgets for them more difficult to detect. Scratch register gadgets are uncommon because they don't persist between function calls, thus the compiler doesn't need to store them to the stack. Such devices will almost certainly only exist in the form of mismatched parses, making them less probable. The implementation of ASLR varies per operating system. By default, Windows 8.1 and Mac OS X randomize everything. Regrettably, both systems only re-randomize system libraries when they reboot. This may result in a BROP-like situation where references to system libraries are leaked. Reboots are uncommon on client and laptop systems, as users pause and resume more often than reboot. As for ASLR, Mac OS X only supports 16 bits of entropy, putting it behind other 64-bit operating systems. The efficacy of ASLR on Linux is determined on the installation and its PIE setup. Ubuntu, for example, does not activate PIE by default, but does so on a case-by-case basis depending on risk.

3. CONCLUSION

The author has concluded about the hacking blind, We demonstrate that, given the proper circumstances, exploits may be written without having any prior knowledge of a particular binary or source code. This is useful for stack vulnerabilities in which the server process resets after crashing. On contemporary 64-bit Linux servers, our approach is capable of defeating ASLR, NX, and stack canaries. Two new methods are presented: generalized stack reading, which overcomes complete ASLR on 64-bit platforms, and the BROP attack, which can locate ROP gadgets remotely. Our completely automated tool, Braille, was tested against actual versions of yaSSL+MySQL and nginx with known vulnerabilities, as well as a fake proprietary service running an unknown binary, and took under 4,000 requests to launch a shell in under 20 minutes. We demonstrate that architectural patterns such as forking servers with numerous worker processes may conflict with ASLR, and that ASLR is only effective when applied to all code segments in a binary (including PIE). Furthermore, security through obscurity, in which the binary is unknown or scrambled, can only delay rather than prevent buffer overflow attacks. To counteract our approach, we recommend that systems rerandomize ASLR and canaries after any crash, and that no library or executable be excluded from ASLR.

REFERENCES

1. Bittau A, Belay A, Mashtizadeh A, Mazières D, Boneh D. Hacking blind. Proc. - IEEE Symp. Secur. Priv., 2014;227–242. doi: 10.1109/SP.2014.22.

2. Erickson J. Hacking: The Art of Exploitation, 2nd Edition. .

3. Jordan T. A Genealogy of Hacking. University of Sussex Word Count: 8,253 (9,554 with Bibliography included). 2017;253:1–34.

4. Cekerevac Z, Dvorak Z, Prigoda L, Cekerevac P. Hacking, protection and the consequences of hacking. Communications - Scientific Letters of the University of Zilina. 2018;18(4):129 – 133. doi: 10.26552/com.C.2018.2.83-87.

5. Jordan T. A genealogy of hacking. Convergence: The International Journal of Research into New Media Technologies, 2017;23(5):28-544. doi: 10.1177/1354856516640710.

6. Omoyiola BO. The Legality of Ethical Hacking. J. Comput. Eng., 2018;20(1):61-63.

7. Barros US, Barros MS. A Survey of Ethical Hacking process and Security. in International Conference on System Modeling & Advancement in Research Trends (SMART), 2015.

8. Billig J, Danilchenko Y, Frank CE. Evaluation of google hacking in Proceedings of the 5th Annual Conference on Information Security Curriculum Development, InfoSecCD '08, 2008, doi: 10.1145/1456625.1456634.

9. Lakshmi C, Basarkod PI. Basics Of Ethical Hacking. 2015.

10. Suresh Kumar VVN. Ethical Hacking and Penetration Testing Strategies. Int. J. Emerg. Technol. Comput. Sci. Electron., 2014;5(3):3389-3393.