HackTheMachine Track 2 Write-up

Theodore Kim

HackTheMachine

HackTheMachine is an annual competition organized by the United States Navy with the purpose of developing a community of interest for digital maritime technologies amongst Department of Defense employees, contractors, and academia. This year, the event took place on September 6, 2019 at the Intrepid Air and Space Museum, and September 7 – 8, 2019 at the NewLab facility in the Brooklyn Navy Yard. This year HackTheMachine, while a single event, consisted of three separate tracks, each focused in a different area of emerging maritime digital technology. The first track, Hack The Ship, dealt with cybersecurity and the evaluation of the digital robustness of common technologies utilized at Naval facilities and on its vessels. Participants had to perform various penetration tests to identify vulnerabilities in five different scenarios and execute attacks to capitalize on those vulnerabilities to cause unintended behaviors, disrupt services, and perform unauthorized actions.

S.P.L.A.S.H.

New to HackTheMachine in 2019 is SPLASH, a scenario which challenged participants to gain unauthorized access to a system by exploiting a publicly exposed process. Specifically, the challenge consisted of a server running a process which was responsible for executing a light flashing pattern on a circuit board. A user was given access to the server and could pass commands which altered the flashing light pattern. The objective was to utilize the user interface provided to execute an unauthorized command and activate a spinning light, mounted behind the server, without disrupting the currently running process. This is a common strategy utilized by malicious actors wishing to infiltrate a system, as often it is the less significant, simpler processes that provide an exploitable vulnerability which can affect the entire system as a whole.

Given the vulnerabilities that the competition organizers built into the process, completing the first task of attaining system control was not difficult. The type of vulnerability introduced to the program (an unchecked string copy) can be turned into a buffer overflow and allows the attacker to change the control flow of the program and execute arbitrary code sections to achieve desired functionality (in this scenario, opening a shell on the host machine). This methodology is referred

to as return-oriented programming, or ROP. The actual difficulty in this scenario, and the core technology being evaluated, was producing an exploit that worked on multiple, diversified binaries. Diversification is a type of assembly-level obfuscation which introduces arbitrary instructions into a compiled binary in order to prevent attackers, who many have access to a vulnerable program's source code, from predicting the location of the code sections which are used in a ROP attack to execute arbitrary commands. Diversification is random and indeterministic, meaning that the content and order of a diversified binary's code cannot be predicted even if the attacker has access to both the original binary and the diversification tool. In production, programs that are deployed onto multiple systems would be diversified such that no two systems would have the exact same binary, hampering attacker's abilities to use a vulnerability found on a single system to exploit multiple hosts.

Executing the Return Oriented Program Attack

The return-oriented programming strategy utilized for the initial attack utilized the fact that the standard C library (libc) was dynamically linked during compilation and was unaffected by the diversification performed on the original binary. The reason for this comes from the behavior of dynamic linking at compile time and runtime. Dynamic linkage is used to reduce the size of a binary by allowing it to utilize a local copy of a library local to the host operating system. The library is loaded into the process memory at runtime and the locations of its functions in its loaded memory segment are matched with local references in the compiled binary. While the local references may change, because dynamically linked libraries are local to the operating system, they are unaltered during diversification. Therefore, the same code segment opening a shell on the target machine (found in the `system` function in libc) would exist at the same memory location across all versions of a binary and could be reliably jumped to using the buffer overflow. The task of producing a unified exploit turned out to be easier than the organizers of the event had anticipated, so they introduced another level of complexity after several teams had completed the previous task of creating a unified exploit.

To solve the issue of libc being unaffected by diversification, the program was statically linked with libc, compiled, and diversified. Static linkage involves explicitly including the library functions in the compiled binary (rather than relying on the operating system to provide the library file then match references in the two files), which could then be obfuscated during diversification.

Furthermore, a minified, less vulnerable version of libc (musl libc) was used for the static compilation which did not include the same exploitable code in its `system` function as the original libc library. A bounty was added to exploit this new system.

Adapting the Original Attack on the Diversified, Statically Linked Binaries

Upon analysis of the new, statically linked, diversified binaries, it seemed as if the diversification did accomplish its goal. While the buffer overflow vulnerability still existed, the code segments used in the previous iteration to initialize the system shell did not. Furthermore, code segments that could be used to exploit the system using the vulnerability that the organizers artificially placed into the process were now located in unpredictable locations in the multiple versions of the binary. Another added feature of this particular diversification utility was the addition of random register clears (i.e. `xor rdi, rdi`) interspersed throughout the binary which prevented multiple code segments from being used in sequence to achieve complex attack functionality. The final challenge introduced by the new binary was the limitation of the buffer overflow: rather than a `memcopy` or `gets`, the overflow existed as a bad `strcopy` call. While `memcopy` or `gets` allows for an arbitrarily long buffer overflow, `strcopy` stops copying data from the source buffer to the destination buffer at the first occurrence of a NULL byte (0x00). Given that 64-bit system addresses often start with at least two NULL bytes, this only allowed for a single address to be overflowed onto the system. This only allowed for a single jumps in the code and severely limits the complexity of possible attacks on the system.

Eventually, solutions to all of the aforementioned challenges were found. To overcome the limited buffer overflow size, a stack pivot was performed which moved the location of the stack into a fixed length buffer which was accessible through the user input. Furthermore, given that a string pointing to the location of the system shell ("/bin/sh") was already included in the binary by the event organizers it was determined that a shell could be opened using the assembly level exec system call rather than the libc `execv` function which was used in the original binary version but was absent from the current version. The new exploit involved a total of six steps:

1. Pivot the stack into the fixed length user input buffer
2. Set rdi point to the "/bin/sh" string
3. Set rsi to 0 (null pointer for environment)

4. Set rax to 0x3B (the syscall code for exec)
5. Set rdx to 0 (null pointer for arguments)
6. Execute syscall

However, even with the aforementioned strategy, executing an exploit on binaries other than the original would be theoretically impossible due to the fact that the code sections which perform the required actions would be at unpredictable locations and could not be reliably jumped to across multiple binary versions.

Further analysis of the provided examples of the diversified binary reveal three key shortcomings of the diversification utility which could be successfully exploited to achieve the aforementioned exploit:

- The diversification did not change the location of global variables (like the "/bin/sh" string), therefore, the location of that pointer could be reliably referenced in every exploit.
- The diversification did not introduce instructions which would alter registers needed to for a function then roll them back later (i.e. if rdi was needed for function A, the diversification utility would not introduce `xor rdi, rdi` until rdi's value was committed, either via a push to the stack or passed to a function).
- The diversification did not touch certain regions of the program which are critical to the execution of the binary by the compiler, particularly libc_start, which is the entry point for the binary and is responsible for passing `main` its arguments and setting the environment variable.

These shortcomings meant that there were certain regions in the binary (usually either low in memory or high in memory) that were consistent across all versions of the binary. Searching these regions for useful code segments produced just enough to achieve the aforementioned exploit actions. Given that they were in regions untouched by diversification, they could be reliably jumped to in the exploit, making it useable for all versions of the diversified binary.

Reproducibility and Recommendations

It is unclear if this exploit means that the diversification utility tested is completely ineffective in preventing ROP attacks. While the implementation of libc_start is usually the same across multiple binaries, the exploit also utilized small code segments from within the binary header and certain critical regions of the statically compiled libc which are not consistent across multiple different binaries (as in multiple programs, not diversified binaries). Without another diversified program, it is difficult to make the determination if this exploit is reproduceable. However, given the attack strategy utilized in this case, it is reasonable to assume that it could be reproduced as long as the attacker has access to at least two diversified versions as similarities between the two binaries could be used to extrapolate larger areas of similarities between other diversified versions.

While it is unlikely the diversifier would be able to alter OS dependent sections like libc_start without potentially preventing proper functionality of the process, it is easy to diversify the globals section by introducing random NULL bytes before and after global variables to alter their location in memory. This would potentially make diversification more complex, but since it is a compile time process, its effect on the program's functionality would be minimal. In the end, diversification is only a tool used to mitigate the exploitability of vulnerable programs and the only true fix would be to eliminate vulnerabilities such as buffer overflows from the original program.