Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 83 (2016) 1301 – 1306

Procedia Computer Science

The International Workshop on Parallel Tasks on High Performance Computing (THPC-2016)

Counter-measures against stack buffer overflows in GNU/Linux operating systems.

Erick Leon[a], Stefan D. Bruda[a]*

[a]*Bishop's University, 2600 College St., Sherbrooke QC, J1M 1Z7, Canada*

Abstract

We address the particular cyber attack technique known as stack buffer overflow in GNU/Linux operating systems, which are widely used in HPC environments. The buffer overflow problem has been around for quite some time and continues to be an ever present issue. We develop a mechanism to successfully detect and react whenever a stack buffer overflow occurs. Our solution requires no compile-time support and so can be applied to any program, including legacy or closed source software for which the source code is not available. This makes it especially useful in HPC environments where given their complexity and scope of the computing system, incidents like overflows might be difficult to detect and react to accordingly.

Keywords: Buffer overflow; Stack; GNU/Linux; ptrace.

1. Introduction

Most HPC systems run some variant of the GNU/Linux operating system, such as the CLUMEQ in Quebec, which uses CentOS 6.3. Tackling security issues in GNU/Linux is therefore a priority for HPC applications. One of the most pressing issues is buffer overflow. Indeed, a study that surveyed Common Vulnerability Scoring System (CVSS) data between 1988 and 2012 showed that buffer overflows were the most often reported vulnerabilities, at roughly 14% of the total amount[1]. Thus, tackling such a recurring problem in such a widely used operating system is a priority for researchers and various industries alike.

In this paper we consider the particular kind of buffer overflows that happens on the stack. The outcome of such

* Corresponding author. Tel.: +1-819-822-9600 ext 2374;
E-mail address: stefan@bruda.ca

doi:10.1016/j.procs.2016.04.270

an overflow is usually an execution error of the program, but if the overflow is performed by a malicious entity, then it can be exploited and can lead to the execution of malicious code.

We develop a solution for detecting stack buffer overflow at run time and without any compile-time support. Our solution is therefore particularly useful for closed source or legacy software, with no available source code. Our system actively analyses an executing program, reacts to the appropriate opcodes, and makes sure that the control flow has not been tampered with. Should a problem be detected, we send a message to the system administrator with pertinent information that can be used for subsequent forensic investigations.

Our system detects all the stack overflow events at the expense of significant overhead on the running time. We therefore offer a faster version, whose functionality is based on empirical observations on code that is likely to be malicious. While we greatly reduce the overhead this way, the faster version is no longer guaranteed to catch all the overflows. Further investigations are needed to assess the effectiveness of this version.

Our solution uses ptrace, a tool which is available in most GNU/Linux distributions, and so should be easily applicable to any such a distribution.

2. Literature review

Buffer overflow has been an actively research area throughout the history of Computer Science. The proposed solutions include both hardware and software approaches. Some offer mechanisms for preventing overflows from occurring, while others react once overflows happen.

Address space layout randomization (ASLR)[2] introduces randomness into the virtual memory layout of a process in order to change the binary mapping of stack memory regions and dynamic library linking before the process executes in order to nullify any attacks that operate under the assumption of known static values.

Stack canaries are values that are inserted into known locations in memory in order to detect buffer overflows due to the fact that when one happens the first data to be corrupted will usually be the canary[3,4,5].

Bounds checking[6] operates by keeping track of the address boundaries for objects, buffers and arrays, and constantly checking the load and store operations that access those resources.

SecureBit2[7] works on top of SecureBit, which introduces a hardware bit to protect the integrity of addresses, while SecureBit2 works to protect SecureBit.

Dynamic information flow tracking (DIFT)[8] tags untrusted data in order to track it through the system. A hierarchical model is put in place so that new data that is derived from untrusted data can be generated.

Chaperon[9] and *Valgrind* are commercial tools that intercept `malloc` and `free` calls directly from a binary executable.

The *Never-execute bit* (Nx-bit)[10] represents a mark on certain areas of memory which makes them non-executable, so that the processor will refuse to execute code within those areas.

One research work[11] proposes a hardware and software hybrid solution to protect against buffer overflows by introducing new assembly functions and a method using boundary checks.
A recent investigation[12] introduces a new instruction to limit the consumption of system resources by the process.

The idea of monitoring the stack during the execution of call and return opcodes was proposed earlier[13,14]. A proof-of-concept system that protects the instruction pointer in 32-bit systems (EIP) from modification using kernel properties was presented.

3. Solution

The stack is a last-in-first-out (LIFO) list that will store the values pushed by the `call` opcodes. These values will then be used by `ret` opcodes to return to the address after the corresponding `call`. Our solution must work during run time and actively detect the overflow whenever it happens. We must be able to work with both 32-bit and 64-bit architectures. Therefore our solution will intercept the appropriate instruction pointers for both architectures (`eip`/`rip` respectively) and whenever a `call` opcode is detected, the next address after it will be stored in a separate buffer that we control. Then when a `ret` is issued, we will verify that the value has not been tampered with, by comparing the value in the stack to the value in the buffer we control. Finally, should a problem be detected, an automatic message will be sent to the system administrator through the network with the pertinent information.

Our solution works using the ptrace system call[15] as the main engine, ptrace is a built-in tool in GNU/Linux operating systems that will allow us to intercept certain resources during the execution of a process for analysis, thus eliminating the need for any extra resources.

3.1. Procedure

In practice, in order to execute the program under the control of our system we launch it like this:

```
./<launch ctl> <target program>
```

where `<launch ctl>` is our control program (that controls the launching of other programs) and `<target program>` is the process to be analyzed. Under the hood we need first to tell the kernel that a process will be traced. This is accomplished with the `PTRACE_TRACEME` call whose syntax is:

```
ptrace(PTRACE_TRACEME, pid,...);
```

In this call `pid` is the process id to trace. This is usually followed by an `execve()` system call which is used to obtain said `pid`. It is worth mentioning that in our code the `pid` is a value input by the user and then sent to a function, so we use:

```
execl(programname, programname,(char*)0);
```

where `execl()` is used to obtain the `pid` which is stored in the variable `programname`. Once `PTRACE_TRACEME` is issued, the kernel is aware that the process is being traced and `ptrace` splits the process into two parts, a parent and a child. This mechanism allows `ptrace` to hand control back and forth from the parent to the child in order to detect or perform operations as needed. In this case, the child will be executing the process and the parent will check for arguments or look into registers. For our construction the child executes the `execve()` system call and so hands control over to the parent. Now we must issue a `wait()` call to stop the execution of the process during the first instruction, we accomplish this in our code with:

```
wait(&wait_status);
```

The parameter `wait_status` is just a variable used to manage the wait signal. It is at this point that we must read the current register values of the process. We accomplish that with `PTRACE_GETREGS` to obtain the registers and then `PTRACE_PEEKTEXT` to obtain the values we need. Our implementation of this in our code is as follows: To obtain the registers we do:

```
ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
```

where `child_pid` is the process being traced. Then to obtain the instruction pointer for our 64-bit platform we use:

```
ptrace(PTRACE_PEEKTEXT, child_pid, regs.rip,0);
```

where `child_pid` is the process being traced and `regs.rip` is the instruction pointer for 64-bit architectures. We use temporary variables to store this value.

At this point in our solution we have successfully stopped the execution of the process during run-time, obtained the value for the instruction pointer, and we must now proceed to see if the current instruction being executed is a `call` or a `ret`. We accomplish this by storing the hexadecimal values of the instruction and evaluating for opcodes. To look for a `call` opcode we then do:

```
if (op==0xe8 || op==0x9a || op==0xff || op2 == 0xff00)
```

The following on the other hand will determine whether the opcode is a `ret` code:

```
if (op==0xc3 || op==0xcb || op==0xc2 || op==0xca)
```

At this stage, we must evaluate each condition accordingly. When a `call` is found, we will obtain the stack pointer and then we store that in a buffer under our control. To obtain the stack pointer for our 64-bit platform we use:

```
ptrace(PTRACE_PEEKDATA, child_pid, regs.rsp, 0);
```

where `child_pid` is the process being traced and `regs.rsp` is the stack pointer for 64-bit architectures.

If on the other hand a `ret` is detected, then we must check whether the return address on the stack has not been modified. To accomplish this we obtain the current return address from the stack and compare it with the latest value stored in our buffer. If both values are the same, then nothing is wrong and the program continues. However, if the values are different, we then compare the current value in the stack to the value next-to-last in our buffer; if they still do not match, we then compare it to the next value in our buffer, and so on. It is important to note that our buffer is effectively a stack, which stores the latest value on the top which is then compared with the value of the normal stack when a `ret` happens.

Once the comparison between the current value on the stack and the values in our buffer is complete, if the current value in the stack is not found then something modified it on the stack and therefore we assume there has been an attack such as a buffer overflow. If this is the case, we now issue a real-time message to the system administrator and kill the process. The message includes the network address of the host, the current system time of the host, the hostname, and the instruction that was executed when the incident occurred (in hexadecimal). While an argument could be made regarding the actual necessity of the message, it is important to remember what we established at the beginning of this document: we seek to develop a solution that not only automatically detects problems, but also issues notifications that contains precise information. In this case the fact that we are sending valuable information like time, hostname, network address and even more so, the actual instruction that caused the problem, can potentially save the user time and other resources should the need to perform forensic analysis arise.

At this point in our solution, we have successfully stopped the execution of the process during run-time, obtained the value for the instruction pointer and evaluated whether the current instruction being executed is a `call` or a `ret`. If a `call` was found then the corresponding value is stored in a buffer, while if a `ret` was found then we compare the current value in the stack to the values stored in the buffer; should a discrepancy be detected, our solution automatically issues a message with pertinent information to the system administrator over the network. It is now necessary to perform this during every instruction executed in order to fully protect the process. To accomplish this, we use `PTRACE_SINGLESTEP` to advance the process to the next instruction. The code for this is:

```
ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0);
```

Finally, the child hands control back to the parent and the parent waits for notification from the kernel with a `wait()` call, thus completing the cycle. This whole sequence will take place during every instruction in the process which translates into a successful, working solution that fulfils the objectives we had set out to accomplish.

3.2. Faster

As stated earlier, we have successfully developed a working solution that accomplishes the objectives we had set out to achieve. However there is still one issue, namely the time it takes to analyze the process. It all boils down to the number of iterations and the use of `ptrace`; while we cannot optimize `ptrace` since it is built into the system, it is still the case that the bigger the process we are analyzing, the larger the number of instructions to inspect and thus, the bigger the buffer will need to be. In practical terms, the time needed to perform our operations will increase accordingly because we are effectively analyzing every single instruction. For these reasons we propose a solution to this issue taking into consideration `write()` system calls to detect stack buffer overflows. This is because in practice, while monitoring the execution of overflowing processes; we detected that system calls were issued at critical points in the buffer overflow exploit. More specifically we identified that such calls were in place when the last `ret` was found (the place of the detection of the overflow). In practical terms, the mechanisms between the normal and fast versions are effectively the same; however the change comes via this piece of code which detects only `write()` system calls:

```
if((regs.orig_rax == SYS_write));
```

4. Testing

In order to test the effectiveness of our system we used the code created by the Mr.Un1k0d3r RingZer0 Team[16] that exploits a buffer overflow to execute a piece of shell code which in turn obtains the password files of a GNU/Linux system. Both our normal and fast solutions successfully detected the overflow, killed the target process and reported it, both on screen and through the network message sent to the system administrator. Given the fundamental steps followed by this exploit, we can safely assume that any stack manipulation will be detected as long as it follows the steps described in the previous section. In fact, in order to prove that our normal solution is ready to handle any such stack modification, we also modified the values in the stack during execution and after the process had been launched. This behavior effectively bypasses counter-measures like ASLR that act once (usually at the start of execution) and assume that the process is therefore safe which in practical terms means that it would go undetected. Our solution however, detected this execution-time manipulation and reported as expected[17].

We also tested GNU/Linux commands commonly used by system administrators as a simple reference for verifying the reliability of our system and also for determining its run-time overhead. The timing measurements were accomplished using the system time (kernel) requirements by issuing the `time` command, which does not represent a measurement of time as we conventionally measure it. For reference, we will use a system with an AMD E2-2000 APU with Radeon(tm) HD Graphics x 2, 7.4 GiB RAM with Fedora 23 64-bit installed on a 120 GB SSD and the partition and root directory fully encrypted with LUKS with all pertinent files located in user directories.

Table 1 Running time

Program	Detection enabled	Time (in seconds)
/usr/bin/cal	No	0.006
	Yes – Full version	29.47
	Yes – Fast version	0.015
/usr/bin/uptime	No	0.007
	Yes – Full version	95.34
	Yes – Fast version	0.026
/usr/bin/w	No	0.049
	Yes – Full version	322.41
	Yes – Fast version	0.196
Vulnerable executable[16]	No	0.006
	Yes – Full version	2.36
	Yes – Fast version	0.115

The results show that while the "Full" version of our solution takes a significant amount of time relative to the other results, it guarantees that every single instruction will be accounted for which is indeed the ideal solution to tackle a problem like stack buffer overflow where the error can potentially be at any instruction. Meanwhile, the fast version is significantly faster but at this stage it would only work under specific conditions and is therefore proof-of-concept.

Finally, in order to verify the robustness of our solution when applied to GNU/Linux as an operating system we also tested it on different distributions. Specifically the OpenSUSE 13.2, Fedora 21 and Ubuntu 14.04 LTS distributions in both their 32-bit and 64-bit versions respectively using virtual machines and also measuring internal clock speeds using the `time.h` standard library of C coded into our solution. The results of these tests and further data can be found at the source documentation[14,17].

5. Conclusions

More details on the system described in this paper are also available[14,17]. The complete source code is also available[14].

Our solution works as a counter-measure against stack buffer overflows in GNU/Linux operating systems. We produce an applicable result which handles both 32-bit and 64-bit environments, as well as operating during execution-time, which is indeed the ideal solution. On top of that, we also provide an alternate solution with the fast version. This fast version is a proof-of-concept code that works under a specific set of conditions yet could potentially be expanded and developed further. Our first solution can be implemented in server environments for deep assembly-code level forensic analysis where every single instruction of an executing process is being monitored, while the fast version, although proof-of-concept for 64-bit architectures, presents an alternative to significantly reduce the time needed for detections. In particular, our system presents a readily available solution to monitor processes in GNU/Linux-based HPC environments without the need to install or include any extra resources which are often required by security applications.

5.1. Discussion

Regarding limitations and time constraints, the performance could be optimized if we were to focus on individual GNU/Linux distributions. We could address each combination of distribution and hardware, and filter out the specific opcode combinations for each. An example of this behavior is the `rex.w + ff` prefix `call` opcode which changes depending on the distribution being used. In 32-bit systems, the `c2` opcode pushes values onto the stack which are handled by certain `eax` values. This behavior will trigger a false positive which we address by skipping any such instance, however we could also strengthen the conditions to intercept this behavior. Currently our system does not work with programs that have a graphical user interface. We believe that the cause of this shortcoming is related to the way threads are generated in the target process. We leave it as an open problem, though it is worth mentioning again that this solution is ready to be implemented in server environments where graphical user interface systems are usually not present.

Compared to other proposed counter-measures against buffer overflows, our solution does not require hardware modifications like the ones proposed by the likes of SecureBit, nor any extra set of libraries, data, or source code like the ones proposed by stack canaries. In addition, we are successfully able to detect incorrect stack manipulations even after the execution has been started; such modifications would bypass countermeasures like ASLR but not our system. Compared with the earlier related work[13] we do not rely on the severely limited kernel memory to back up stack pointers, though this is accomplished at the expense of increasing run-time overhead; we thus believe that the faster variant is worth developing in depth as a good compromise between security and overhead. Finally, our counter-measure utilizes mechanisms that are not only found in GNU/Linux systems, but in most Unix-based systems as well. As such, we believe that our work could be ported over to other Unix-based operating systems. This is relevant to HPC environments that may use other unix-based operating systems.

References

1. P. Roberts, At the Vulnerability Oscars, The Winner Is... Buffer Overflow!!, https://www.veracode.com/blog/2013/02/at-thevulnerability-oscars-the-winner-is-buffer-overflow, retrieved April 2015.
2. Address Space Layout Randomization, (Mar.2003), https://pax.grsecurity.net/docs/aslr.txt, retrieved Feb. 2015.
3. G. Duarte. Epilogues, Canaries, and Buffer Overflows, (Mar. 19 2014), http://duartes.org/gustavo/blog/post/epiloguescanaries-buffer-overflows/ , retrieved Feb. 2015.
4. J. Deckard, Defeating Overflow Attacks, GSEC Practical Assignment, Version 1.4b Option 1, SANS Institute InfoSec Reading Room (April. 14,2004),http://www.sans.org/readingroom/whitepapers/securecode/defeating-overflowattacks-1403.
5.P. Silberman and R. Johnson, A Comparison of Buffer Overflow Prevention Implementations and Weaknesses, presentation at Black Hat USA, Caesar's Palace, Las Vegas, NV, USA (Jul. 2004).
6. W. Chuang, S. Narayanasamy, B. Calder and R.Jhala, Bounds Checking with Taint-Based Analysis, in Proceedings of the 2 nd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'07), Ghent, Belgium (Jan. 2007), pp 71-86.
7. K. Piromsopa and R. J. Enbody, Secure Bit2: Transparent, Hardware Buffer-Overflow Protection, Michigan State University,.
8. M. Dalton, H. Kannan and C. Kozyrakis, Real-World Buffer Overflow Protection for Userspace & Kernelspace, in Proceedins of the 17 th USENIX Security Symposium (USENIX Security '08), San Jose, California, USA (Jul. 2008)
9. M. Zhivich, T. Leek and R. Lippmann, Dynamic Buffer Overflow Detection, in Proceedings of the 2005 Workshop on the Evaluation of Software Defect Detection Tools, Chicago, IL, USA (June 12, 2005.
10. Non-Executable Pages Design and Implementation, (May 2003), http://pax.grsecurity.net/docs/noexec.txt, retrievedFeb. 2015.
11. Z. Shao, C. Xue, Q. Zhuge, E. H.-M. Sha, Security Protection and Checking in Embedded System Integration Against Buffer Overflow Attacks, in Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2004), Volume I, Apr. 2004, pp. 409-413.
12. Z. Shao, J. Cao, K. C. C. Chan, C. Xue, E. H.- M.Sha, Hardware/Software Optimization for Array & Pointer Boundary Checking Against Buffer Overflow Attacks, Journal of Parallel and Distributed Computing, 66:9, 2006, pp. 1129-1136.
13.B. Teissier and S. D. Bruda, Toward Preventing Stack Overflow Using Kernel Properties, in Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOFT-EA, 2014), Vienna, Austria (August 2014), pp. 369-377.
14. Run-time Solutions to the Stack Overflow Problem, https://part.bruda.ca/papers/run-time_solutions_to_the_stack_overflow_problem
15. ptrace(2) – Linux man page, http://linux.die.net/man/2/ptrace, retrieved Feb. 2015.
16. Mr. Un1k0d3r RingZer0 Team, 64 Bits Linux Stack Buffer Overflow, http://www.exploitdb.com/wp-content/themes/exploit/docs/33698.pdf,retrieved Feb. 2015.
17. E. Leon, The “ptrace” Solution to Stack Integrity Attacks in GNU/Linux Systems, M.Sc.Thesis, Bishop’s University, Sherbrooke, Quebec,Canada, May 2015.