INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 4.295
(Volume 4, Issue 6)
Available online at: www.ijariit.com

Buffer overflow: Mechanism and countermeasures

Vikrant Chatole
vikrant.chatole@adypu.edu.in
Ajeenkya D.Y. Patil University, Pune, Maharashtra

Gauransh Nagar
gauransh.nagar@adypu.edu.in
Ajeenkya D.Y. Patil University, Pune, Maharashtra

ABSTRACT

The invention of Computers, Information Technology and thence Internet has led humanity to a new era of revolution. We, as humans, have stored more data in the last 20 years than the whole of human history. In May 2018 Forbes announced that we have created 90% of data all data in the past two years. That describes the way information storage and usage is picking up the pace. But are our basic pillars of storing data and processing full proof and completely secure? Buffer Overflow is currently the most hostile vulnerability in the basics of information storage and processing of our computing technology. The paper discusses this vulnerability in thorough details. Ways systems are coping up with this and methods used to overcome this vulnerability present in the basics of our most important invention.

Keywords— *Buffer overflow, Ethical hacking, Information security*

1. INTRODUCTION

Buffer overflows makes up an extensive portion of all security attacks over the globe. The characteristics that make buffer overflows so threatening is that it runs inside the target program with all its privileges [1]. Almost any kind of information can be injected if the attacker knows the address spaces. Since the Buffer overflow attacks work very close to the hardware, with very little information a humongous damage can be done.

The mechanism involves injecting malicious code next to an array, per se an array that is where the 'buffer' in the name comes from, which we discuss in detail in section 2. Counter measures to overcome this vulnerability will be explained in section 3. Section 5 presents our conclusion.

2. THE ATTACK MECHANISM

The root of buffer overflow exists in the way the registers store and process information [5]. A variety of registers work together in order to process to process information. Mainly, as mentioned below.

- **EAX**: Extended Accumulator Register
- **EBX**: Extended Base Register
- **ECX**: Extended Counter Register
- **EDX**: Extended Data Register
- **ESI**: Extended Source Index
- **EDI**: Extended Destination Index
- **Flags** (special register)
- **ESP**: Extended Stack Pointer
  Marks the top on the stack
- **EBP**: Extended Base Pointer
  Points to the base address of the stack.
- **EIP:** Extended Instruction Pointer (special register, read-only)
  Contains address of the next instruction of the program.
  Points always to the "program code" memory segment

Figure 1 shows the growth of the stack that goes from ESP towards EBP. **RET** stands for "**Ret**urn from procedure" and it stores the address to jump to after the function finishes i.e. the address of the ESP of the next stack to be processed.

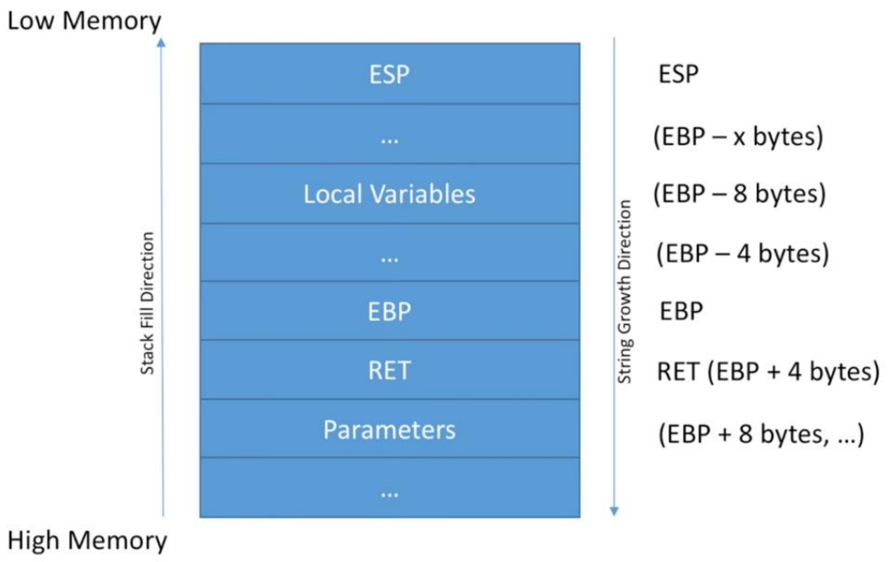

Fig. 1: Growth of the stack

The function present at the address space pointed by the **RET** will be executed next. Each such stack is reserved for every function, array or any sequential logical structure- stack, queue, etc. The loophole exists in the bounds of these. As the stack continue new information is being added down the stack leading to the growth of the stack but if the bounds aren't checked, the stack can grow beyond the reserved memory, leading to the registers like RET getting overwritten. If the RET is overwritten with any other information, the stack won't be pointing to the next address space for continuous execution and program crashes. Figure 2 explains this pictographic detail and simplicity.

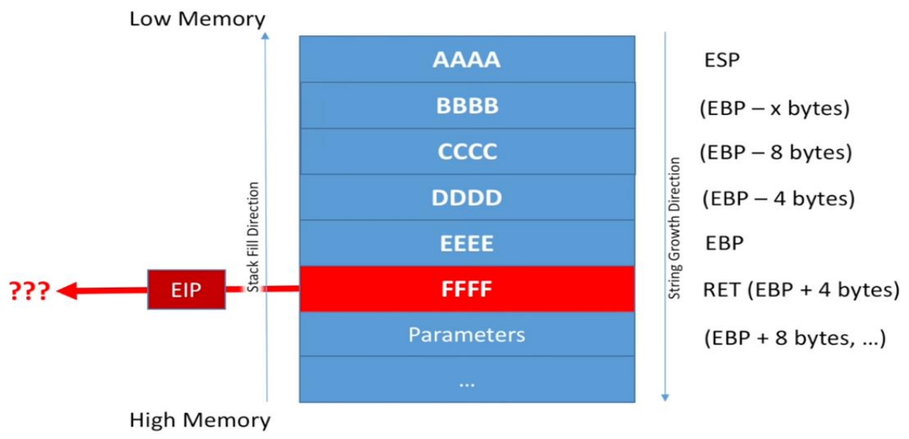

Fig. 2: Pictographic detail and simplicity of the stack

As illustrated in Figure 2, the stack continues to grow (stored information denoted as repeated alphabets) beyond the point of reserved storage and overwrites the RET, the program has no further instructions to execute and thence leading to crash. This could be disastrous for large-scale organisations. This vulnerability has no outsize limit, any size of the program could be brought down with just one extra byte of data in the most vulnerable and unexpected place. But this is not the worst-case scenario. Situations could get even severe from here.

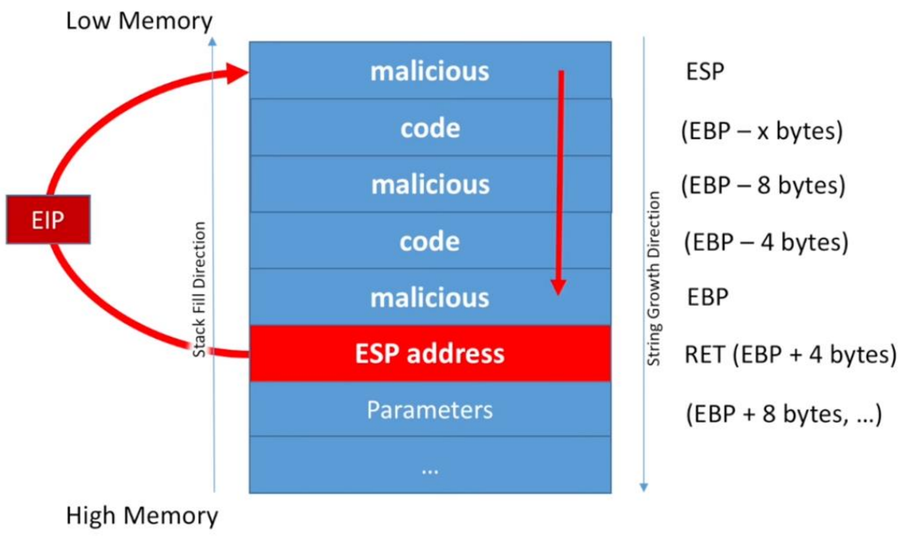

Fig. 3: Process in depth and simplicity of the stack

Imagine a programmer being aware of the exact address pointed by the RET and the bounds of reserved space from the respective stack. That programmer could overwrite the stack to the point of RET being overwritten and replace the pointer address by the address of a malicious function. This will effectively lead the program to execute the malicious function resulting in the programmer even taking the total control of the system. This works locally as well as remotely, which implies that this vulnerability could potentially hand the control of any system to any unauthorised user. Figure 3 describes this process in depth and simplicity.

3. COUNTER MEASURES AGAINST BUFFER OVERFLOWING

3.1 Pointer protection

Pointers are the bases of buffer overflows. Applying constraints over pointers in a program effectively reduces its possibility to be used to overflow the buffer. 'PointGuard' is one such attempt made in this direction. It was first proposed as a compiler extension to avert attackers from manipulating pointers by adding code before and after XOR-encode pointer automatically. PointGuard was never released albeit Microsoft incorporated a similar feature in Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1[6]. They added an API routine at programmer's freedom this improved the performance multiple folds compared to the original Pointer Guard as the API was called only on the programmer's discretion rather than running in the background all the time. This certainly improved the performance but increased the stress over the programmer as to where to invoke the API routine.

3.2 Executable Space Protection:

The idea behind Executable Space Protection (ESP) is that if execution of code on stack or heap could be prevented then the attacker cannot insert the arbitrary or any malicious code in the first place, an attempt to do so will cause an exception to occur [7].

Executable Space Protection is now shipped built-in with CPUs and Operating Systems. Some CPUs feature this technology as NX ("No eXecute") or XD ("eXecutable Disabled") [7]. These features work in unification with software to mark pages with the codes to run with stacks as writable and readable but not executable. Some Unix operating systems like Open BSD and MacOS deliver this from within the OS as a feature while add-ons can be added on other Unix devices. Some of those add-ons include PaX[4], Exec Shield and Openwall. Microsoft supports ESP with the name 'Data Execution Prevention'. Windows addons include BufferShield and StackDefender.

3.3 Address space layout randomization

ASLR or Address Space Layout Randomization arranges the positions of crucial data areas such as the position of the heap, stack, and libraries in a random order in a process' address space [8]. This isn't a good solution because the exact same order can be tailored programmatically by the attacker and buffer could again be exploited.

3.4 Choice of programming language

The language that the program is created in plays a crucial role in the prevention of buffer overflow. In older languages like Fortran, C, C++, and Assembly Languages bounds wasn't an available feature for a long time. These languages are also the most important ones since these work very close to the hardware and empowers the programmer to control memory and address spaces programmatically. Functions like gets(), sprint(), strcpy() and strcat() should be avoided at all cost. These functions do not have the feature of the bound check. Modern Programming languages like Java and C# automatically does a bound check before loading data into the buffer and throws an exception if the data is found to be exceeding the buffer bound. Even though the C and C++ provide techniques to avoid buffer overflow its upto the programmer to incorporate those and are not autonomous and hence programs involving such a risk should be created in modern programming languages like C# and Java.

3.5 Testing

Continuous monitoring of the system's memory resources is very crucial and can stop computing disaster from happening. Most recent big exploits discovered was in 2014, when a vulnerability was discovered to be existing in the Linux Kernel. Linux, which is hailed as one of the most secure OS through the globe was also vulnerable to overflow that granted root access remotely to an unauthorised user. This was known as the Dirty.COW attack.

4. CONCLUSION

As aforementioned buffer overflow is undeniably one the most hostile vulnerabilities present in our systems but it is also the most researched and studied vulnerability. More than 30 years of research has gone into avoiding the overflows, making it less contemporary compared to the severity it once had. Albeit it has increased the significance of monitoring even more, as seen in the Dirty.COW attack even the most secure could be internally vulnerable to overflow and that goes to show how fundamental buffer overflows are.

5. REFERENCES

[1] Michele Crabb. Curmudgeon's Executive Summary. In Michele Crabb, editor, The SANS Network Security Digest. SANS, 1997. Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, Jean Chouanard

[2] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology, Buffer Overflows Attacks and Defenses for the Vulnerability of the Decade.

[3] Buffer Overflow- https://en.wikipedia.org/wiki/Buffer_overflow

[4] PaX - https://en.wikipedia.org/wiki/PaX

[5] Buffer Overflow- https://en.wikipedia.org/wiki/Buffer_overflow#cite_note-35

[6] USENIX Association, Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA August 4–8, 2003, https://www.usenix.org/legacy/event/sec03/tech/full_papers/cowan/cowan.pdf

[7] Executable space protection, https://en.wikipedia.org/wiki/Executable_space_protection
[8] Address space layout randomization, https://en.wikipedia.org/wiki/Address_space_layout_randomization