# Buffer Overflow Exploits

Khurram Farooq and Ayaz Uqaili
SZABIST
Karachi, Pakistan

***Abstract:***

*The focus of this Study is on providing an understanding of buffer overflows, the ways they are exploited, and ways to prevent attackers from abusing them. Although this problem has been around for decades, the devastating effects have been downplayed by the commercial organizations due to the fact that they require a lot of effort to trace and to fix. This has led to a flood of software on the market which claims to be secure, yet can be exploited by wily hackers. As our reliance on closed-source and proprietary systems increases, we have to face the facts that there could be a myriad of security vulnerabilities in the very tools we use to protect critical data. To be informed is to be better armed.*

## 1 INTRODUCTION

Buffer overflows have been causing serious security problems for decades. In the last few years the underlying cause of the majority of computer system and network exploits and vulnerabilities have been the buffer overflow condition. As such, this represents or should represent a top security concern for all entities associated with information security.

The main objectives of this study are

- To understand the nature of Buffer Overflow Exploits.
- To get a gist of the motivation behind such attacks, and the devastation they can cause.
- To gain an understanding of the structure of BOF Exploits and how to find them.
- To understand ways to avoid introducing BOF's in code, and fix existing BOF's.

### 1.1 Why Buffer Overflows?

Buffer overflows were relatively unheard of in the wild before 1996. The landmark that sparked off an interest in buffer overflows was AlephOne's article "Smashing the Stack for Fun & Profit", which appeared in the Phrack of November 1996 [1]. As a result, since 1997, there has been a steady increase in the number of buffer overflows discovered and exploited. It is this steady increase that is the most alarming trend in the security industry. Although buffer overflows have been around for more than a decade, they still continue to be one of the most prevailing security holes in the most hardened and respected Operating Systems. As of May 12, 2003, there are 13 advisories issued by CERT [2], 9 of which are related to buffer overflow vulnerabilities. Further, two of the most damaging worms, CodeRed and SQL Slammer, both spread by exploiting buffer overflows in IIS and SQL Server respectively.

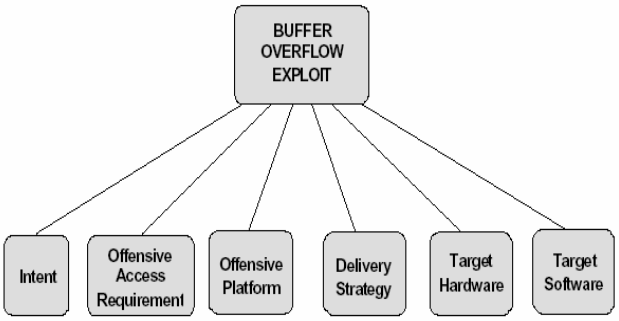
## 2 A THEORETICAL OVERVIEW

### 2.1 Description of a Buffer Overflow

Simplistically speaking, a buffer overflow occurs when the program tries to write more data to a memory location than the space that has been allocated for it. This results in the adjacent data structures getting corrupted. It may result in a core dump, or if it is due to an attack by a hacker, it could lead to execution of arbitrary code.

### 2.2 A Taxonomy of Buffer Overflow Exploits

It is a practice in sciences that everything is categorized and classified. Computer sciences are no different in this respect. If we are to gain a complete understanding of buffer overflow exploits, we must be able to classify them into different types. This requires a sound taxonomy. A taxonomy is defined as "a system of classification that allows one to uniquely identify something". A taxonomy presented by Terry Bruce Gillette [3] to identify Buffer Overflow Exploits has the following categories:




### 2.3 Structure of a Buffer Overflow Exploit

There are many variants of buffer overflow exploits. It is

beyond the scope of this independent study to mention and elaborate each of these types separately. But from what has been observed, all buffer overflow exploits have a few things in common. These are the things which are the essential ingredients of a buffer overflow exploit, and it is necessary for all of them to be present in order to execute a successful buffer overflow exploit.

#### *2.3.1 Victim Application*

This is the application that is under attack. This application accepts user input and writes it to memory without proper bounds checking.

#### *2.3.2 Overflow String*

This is the string that is supplied by the attacker and written by the victim application, corrupting adjacent data structures.

#### *2.3.3 Shellcode*

This is the "arbitrary code" that is to be executed once the application has been successfully exploited. Most of the time it returns a shell to the attacker, hence the name "shellcode".

## 3 PRACTICAL APPLICATIONS

### 3.1 Discovering Buffer Overflows

Discovery of buffer overflows is a very important activity from the point of view of attackers, as well as from the point of view of software developers. The attackers can use these methods to find out new buffer overflows, and write 0-day exploits for them. This will give them an edge over others and will open up a whole new vista of machines that are waiting to be hacked. For the software developers, such techniques provide them with ways to test the applications they have developed for buffer overflows. Needless to say that a buffer overflow in a critical service or a security application is very dangerous, and it is better that it is discovered by the QA department, than if it is discovered by a hacker with less honorable intentions.

The major methods for finding Buffer Overflows in code are:

#### *3.1.1 Manual Methods*

##### *3.1.1.1 Source Code Perusal*

3.1.1.1.1 With Original Source Code

The person wishing to find a buffer overflow manually goes through the source code, looking for occurrences of unsafe programming practices in the code.

3.1.1.1.2 With Disassembled Code

This is the same as above, the only difference is that the original source code is not available, and so disassembled code is used instead.

##### *3.1.1.2 Hit-and-Trial*

In this method, the attacker tries a variety of inputs in different locations in the program. Any erroneous output could be a potentially exploitable hole.

#### *3.1.2 Automatic Methods*

Not much work has been done in automatically searching for buffer overflows in a binary image. But, this is a very important area since we rely on mostly closed-source and proprietary systems for protecting our critical resources. Terry Bruce Gillette proposed a novel method of discovering potential holes in a binary image [3].

The method they propose is that they build up compiled "signatures" of a few dangerous functions which they have already identified, such as printf(), gets(), etc. Then they scan the binary image of the target application, and report the results. The output shows a number of starting points for checking for buffer overflows.

### 3.2 Fixing Buffer Overflows & Best Practices

Most of the buffer overflow conditions are hard to find. If they are found, they are difficult to exploit. If they are exploited, then it is difficult to control the execution path. But this does not mean that we start treating buffer overflows mildly, as has been the case in the past. What is needed is an approach that attempts to minimize the instances of buffer overflows found in the wild. The following advice is a starting point to avoiding buffer overflows:

#### *3.2.1 Advice for Developers*

##### *3.2.1.1 Avoiding Dangerous Functions*

There are a handful of dangerous functions which are prone to buffer overflows. The developer should take care not to use these functions in code. If it is necessary to use these functions, then proper bound checking must be implemented.

##### *3.2.1.2 Input Sanitization*

Any external input, whether from a user or a file, or any other means should be considered hostile. This is one of the most critical yet most overlooked aspects of secure code.

##### *3.2.1.3 Code Review Utilities*

Numerous code-reviewing utilities are available, which

minimize the risk of buffer overflows slipping through the cracks. The developer should benefit from these utilities so that his code is more secure.

#### *3.2.2 Advice for End Users*

End users are pretty much helpless in the amount of work they can do to prevent buffer overflows from taking place. There are some operating systems that disallow execution from the stack, but this is not a very reliable option, since it only eliminates stack overflows, and can be bypassed in some cases.

Barnaby Jack (dark spyrit) [4] has also described an interesting technique to patch programs for which source code is not available. He uses the example of Seattle Labs Mail Server, detects a buffer overflow, exploits it, and describes a way to patch the executable. This technique could be applied to other software, but requires expertise which is well beyond the average end user.

### 3.3 Avenues for Further Research

Despite the fact that buffer overflows have been around since the beginning of computer systems, there has been a comparatively small amount of research in this area. The following are the areas that have been identified where further research can be carried out:

#### *3.3.1 Preventing Buffer Overflows*

This should be one of the top priority areas for software vendors, so that they can protect their consumers from the effects of buffer overflows.

##### *3.3.1.1 Testing Tools*

Testing tools can be developed which eliminate, or at the very least detect buffer overflows.

##### *3.3.1.2 Testing Standards*

Research can be carried out in the area of Quality Assurance, and proper test cases as well as best practices can be documented to ensure that the software under consideration does not contain buffer overflows.

##### *3.3.1.3 New Languages*

New languages can be developed which provide inherent bounds checking, such as Java, so that there is no need to worry about buffer overflows if a developer uses that language to code an application.

#### *3.3.2 Fixing Buffer Overflows*

Since not everyone is up to date with the latest software, there is a need to devise a method to fix buffer overflows.

##### *3.3.2.1 Patches*

Fixing a single buffer overflow does not require a completely new version. Post release fixes will do just fine.

##### *3.3.2.2 Utilities to Prevent Buffer Overflows*

Utilities are available which, when installed, attempt to prevent buffer overflows by different techniques, such as by turning off stack execution, or by analyzing code. These can be researched and identified.

##### *3.3.2.3 Patching Proprietary Software at Home*

As already mentioned, Barnaby Jack's article [4] is a complete step-by-step guide on patching proprietary binaries. But the expertise required to do such a thing is not trivial. Further research on this area could build upon Barnaby Jack's work, and find ways which would bring down the level of expertise required to patch such software.

#### *3.3.3 Discovering Buffer Overflows*

There is a dire need for all buffer overflows in the wild to be discovered, so they can be patched. Research work in this area would be beneficial to the user community in general, since it would reduce the number of incidents talking place.

##### *3.3.3.1 Defining Standards*

Research can be carried out to create an authentic document, which can always be referred to when creating test cases for software. This document would be comprehensive enough to detect all types of buffer overflows, and can be incorporated into the policies of a lot of software houses.

##### *3.3.3.2 Automated Discovery Techniques*

This is a very poorly researched area. We have presented one example of an algorithm that can automatically detect buffer overflows, but it is not very useful. What is needed is an algorithm that does not identify only buffer overflows, but rather it identifies exploitable buffer overflows.

#### *3.3.4 Exploiting Buffer Overflows*

This area of research is not entirely useful, but it is a fact that

has to be faced. Unfortunately, most of the research that is being done is in this field, as can be observed by browsing to any security related site.

#### *3.3.4.1 Plug-and-Play shellcode*

The ultimate dream of a hacker is a universal shellcode, which works on all operating systems, under all conditions, with maximum devastation. This is the aim of the KungFoo project [5], which tries to create a shellcode that is as generic and as universal as possible.

## 4 CONCLUSION

The devastation of Buffer Overflows is mostly due to the adherence to the Von Neumann architecture, where data and code are placed in the same memory. As long as such an architecture is being followed, Stack Overflows can and will be exploited. It is to be kept in mind that the most common type of buffer overflows, and the ones which we have focused, are Stack Overflows. With the advent of new languages that implement strict bounds checking, and work on an object-oriented paradigm, coupled with non-executable pages in memory, Stack Overflows and shellcode will become a thing of the past. Granted, there will be plenty of legacy code that would be exploited with Stack Overflows, but the new frontier awaiting the hacker is the world of Heap Overflows. Here, the hacker will not insert malicious code; instead he will attempt to subvert the logic of the program itself (e.g. mblnAuthenticated=True instead of mblnAuthenticated=False).

It is high time that the computer community wakes up and realizes that "Security through Obscurity" is not an issue. With the Open Source movement in full swing, it is time that major software vendors start releasing source code for their operating systems, as this will allow more experts to go through this code, and make it more secure. If not, then there are many malicious people with a debugger and a disassembler, waiting to pounce upon the unwary user. The code can be encrypted, or obfuscated, but for the machine to execute it, it has to comprehend it. And if the machine can comprehend it, so can the hacker…

## REFERENCES


[1] Elias Levy (AlephOne). "*Smashing the Stack for Fun and Profit*". Online, Phrack Online, Volume 7, Issue 49, File 14 of 16, Available: http://www.phrack.org/, November 9 1996.

[2] CERT Advisories. http://www.cert.org/

[3] Terry Bruce Gillette, *"A Unique Examination of the Buffer Overflow Condition"*, http://www.cs.fit.edu/~tr/cs-2002-12.pdf, May 2002.

[4] Barnaby Jack, aka (dark spyrit), *"Win32 Buffer Overflows (Location, Exploitation, and Prevention)"*, Online. Phrack Online. Volume 9, Issue 55, File 15 of 19. Available: http://www.phrack.org/, September 9, 1999.

[5] Kungfoo Project. http://www.harmonysecurity.com/kungfoo.html