Sri Lanka Institute of Information Technology

Emerging Topics in Cyber Security

Assignment 3

MS19807140

Dimuth Hettiarachchi

Contact No: 0779212607

Created on: 25/11/2020

Contents

1. Buffer Overflow / Buffer Overrun

Buffer overruns in lower-level languages have long been recognized as a concern. The main issue is to mix up users' data and information on program flow control for success and to allow direct access to application memory in low-level languages. C++ and C are the two utmost common buffer overrun languages.

Strictly, when a program requires response to write past the finale of the assigned buffer, a buffer overrun occurs, but there are many similar problems that have often the same impact. Another concern is whether an assailant is permitted to write to arbitrary location of memory outside the program array.

C is the most widely used language for constructing buffer overruns, followed closely by C++. When writing into an assembler, it is simple to build buffer overruns because there are no guarantees. Even though C++ is ultimately as unsafe as C, as a superset of C, it can significantly reduce the capacity to misuse consuming the Standard Template Library (STL) with attention.

Strings and vectors will dramatically minimize errors instead of static arrays and many of the errors result in crashes that are not being exploited. A programmer can prevent mistakes with the increased strictness of the C++ compiler. Our recommendation is to use the C++ compiler to clean up the code even if you write pure C code.

1.1 CWE references

- CWE-121: Stack-based Buffer Overflow
- CWE-128: Wrap-around Error
- CWE-131: Incorrect Calculation of Buffer Size
- CWE-124: Boundary Beginning Violation ('Buffer Underwrite')
- CWE-123: Write-what-where Condition
- CWE-122: Heap-based Buffer Overflow
- CWE-125: Out-of-bounds Read
- CWE-466: Return of Pointer Value Outside of Expected Range
- CWE-129: Unchecked Array Indexing
- CWE-193: Off-by-one Error

1.2 Error explained

```
#include <stdio.h>
void DontDoThis(char* input)
{
	char buf[16];
	strcpy(buf, input);
	printf("%s\n", buf);
}
int main(int argc, char* argv[])
{
	// So we're not checking arguments
	// What do you expect from an app that uses strcpy?
	DontDoThis(argv[1]);
	return 0;
}
```

Let's now compile the application to see what's going on. The novelist used a publication with debugging symbols permitted and check stack deactivated for this demonstration. A good compiler wants to lined up a function that is as slight as DontDoThis, particularly when it is called only on one occasion. This is how the stack looks on its framework just before strcpy is called:

```
0x0012FEC0  c8 fe 12 00  Èþ.. <- address of the buf argument
0x0012FEC4  c4 18 32 00  Ä.2. <- address of the input argument
0x0012FEC8  d0 fe 12 00  Ðþ.. <- start of buf
0x0012FECC  04 80 40 00  .□ @.
0x0012FED0  e7 02 3f 4f  ç.?O
0x0012FED4  66 00 00 00  f... <- end of buf
0x0012FED8  e4 fe 12 00  äþ.. <- contents of EBP register
0x0012FEDC  3f 10 40 00  ?.@. <- return address
0x0012FEE0  c4 18 32 00  Ä.2. <- address of argument to DontDoThis
0x0012FEE4  c0 ff 12 00  Àÿ..
0x0012FEE8  10 13 40 00  ..@. <- address main() will return to
```

Note, all the stack values are reverse. This instance is from an Intel 32-bit machine. This means the least relevant byte of a assessment originates first, so when you look at a memory return address as "3f104000," it's 0x0040103f.

Look at what happens when you overwrite your buffer. The stuffing of the Extended Base Pointer (EBP) Register are the first control information on the stack. The frame pointer comprises the EBP, and EBP shall be cut off if an off-by-one overflow occurs. The program jumps into that position and executes the code given by the intruder when the attacker can access the memory at 0x0002fe 00 (off-by-one zeros out byte).

The afterward item to go is to return the address if the overrun is not limited to one byte. If the attacker is able to manipulate this value and position enough mounting in a buffer to know the spot, you can look at a classic buffer overrun. Notice that you don't have to put the assembling code in the buffer that's overrun (often called the shell code since the utmost popular exploit is the command shell). This is a classic example, but generally speaking, the arbitrary code put in your attacker can be found elsewhere in your software. Do not feel comfortable with thinking the overshoot is limited to a specific area.

1.3 Secure Coding Practices to avoid buffer overrun

1) Replace Dangerous String Handling Functions
2) Check Loops and Array Accesses
3) Audit Allocations
4) Replace C String Buffers with C++ Strings
5) Use Analysis Tools
6) Replace Static Arrays with STL Containers
7) Apply Stack Protection

2. Format String Vulnerability

The root cause of the format string bugs, as with many security issues, is user-supporting feedback deprived of validation. In C++/C, string bugs can be used for writing to random memory positions and the dangerous thing to do is to do so without destroying adjacent memory chunks. This advanced capability enables an attacker to circumvent stack safeguards and even to alter very small memory pieces. The difficult can also arise when an unreliable position that the attacker manages reads format strings. The latter part of the issue is more popular in Linux systems and UNIX systems. Application string boards are typically on Windows systems keeps Dynamic Connection Libraries runnable inside the software or resource (DLLs). If attackers are able to rewrite the key executable or resource DLLs, attaches can attack even easier than string bugs since they can simply modify the code you run.

When Address Space Randomization (ASLR) is implemented, certain attacks cannot be carried out with confidence without a data leak. Since a format strings bug can leak information of the address arrangement of the application, it may lead to a previously defective attack. An additional issue is that unsuitable format requirements for different sized types will trigger either data truncation or write only portions of the value, as we shift applications from 32-bit worlds to 64-bit.

2.1 CWE References

- CWE-134: Uncontrolled Format String

2.2 Error explained

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  if(argc > 1)
    printf(argv[1]);

  return 0;
}
```

Display or storage data formatting can be quite a tough process. Many programming languages therefore have data reformatting routines. In many languages, a string, called the format string, is used to describe formatting information. In reality, a format string is described using a limited language for processing data to make output formats simple to represent. But several developers make a basic mistake—they use insecure user data as a format string. This helps attackers to write strings to create several difficulties.

For now, let's say that it is time before you find out how to make your code run if you allow an attacker to manipulate the format string on a C++/C programme. One particularly disgusting side of this kind of attack is that they can test the stack and precise the attack on the fly before the attack is launched. Indeed the author used a Command Line Interpreter other than the one he used to generate the example for the first time that this attack was made publicly and it did not succeed. Due to the specific versatility of the attack, the sample implementation could be corrected and used by the audience.

2.3 Secure Coding Practices to avoid Formal String Vulnerabilities

1) Never to directly transfer user involvement to a formatting function, and also ensure that the formatted output is processed at every stage.

2) The newer version of Microsoft CRT disables the "%n" specifier but set_printf_count_output allows it to be turned on. The following compilation choices are helpful if you use the gcc compiler:

   - ***Wall***—Enables all warnings, noisy, but produces highest-quality code.
   - ***Wno-format-extra-args***—Checks that the count of arguments is not larger than the number of specifiers.
   - ***Wformat-nonliteral***—Warns if the format string is not a literal and there are no additional arguments.
   - ***Wformat***—Checks to ensure that format specifier arguments make sense.
   - ***Wformat-security***—Warns if the format string is not a literal and there are no additional arguments. Currently, this is a subset of –Wformat-nonliteral.
   - ***Wformat=2***—Enables –Wformat plus format checks not included in –Wformat. Currently equivalent to Wformat, Wformat-nonliteral, Wformat-security, and Wformat-y2k combined.

3. Integer Overflow

Integer overflows, underflows, and arithmetic overflows of any sort have been a problem since computer programming was first launched, especially floating point errors. Integer overflows were subject to security research after heap operations effectively substituted for the basic stack-shattering attacks. Although integer overflows have long been used in exploits, they are now at the root of many recorded problems in recent years.

The key issue is that there are processes where the outcome is not what you will get with the paper and pencil for almost any binary format in which we can pick numbers. In certain languages, there are exceptions, but variable integer types are not usual and overhead.

3.1 CWE References

- CWE-682: Incorrect Calculation
- CWE-192: Integer Coercion Error
- CWE-191: Integer Underflow (Wrap or Wraparound)
- CWE-190: Integer Overflow or Wraparound

3.2 Error explained

```
template <typename T>
void WhatIsIt(T value)
{
	if((T)-1 < 0)
		printf("Signed");
	else
		printf("Unsigned");

	printf(" - %d bits\n", sizeof(T)*8);
}
```

The consequences of integer errors vary from crashes and logic errors to privilege escalation and arbitrary code execution. An attacker has an ongoing incarnation that causes an application to make mistakes in deciding the size of the assignment;

Overflowing a heap. The error can vary from an underspend to zero bytes. You may think you are resistant to integer overflows, but this may be an error if you normally create in a language other than C/C++. Some years ago in Network File System (NFS) logical errors associated with the truncation of integers triggered a bug, with any user accessing files as root. Integer issues were just as severe as catastrophic spacecraft failures.

3.3 Secure Coding Practices to avoid Integer Overflow

1) Do the Math
2) Write Out Casts
3) Don’t Use Tricks
4) Use SafeInt
5) When you use gcc, the –ftrapv option can be compiled. This catches signed integer overflows, but only works for signed integer by calling into different executable functions.

4. C++ Catastrophes

One of the newer types of attacks is errors in C++. Usually, the actual mechanism of attack is one of two variants on the same subject. The first class is that a feature pointer may be used. A lot of usable pointers are transferred by Microsoft Windows, MacOS and X Window System APIs, and C++ is a common means of working with GUI (graphical). Code on the UI. If you can alter the flow of a program with a function pointer.

The second attack relies on a C++ class that includes a virtual function pointer table with one or more virtual methods (vtable). If you can overwrite the stuffing of the class, the pointer to the table will be modified and the code of the attacker will be executed immediately.

4.1 CWE References

- CWE-703: Failure to Handle Exceptional Conditions
- CWE-457: Use of Uninitialized Variable
- CWE-416: Use After Free
- CWE-404: Improper Resource Shutdown or Release
- CWE-415: Double Free

4.2 Error instances

1) Lack of Re-initialization
2) Ignorance of STL
3) Pointer Initialization

4.3 Secure Coding Practices to avoid C++ Catastrophes

1) Mismatched new and delete
2) Re-initialization
3) Copy Constructor
4) Uninitialized Pointer
5) Constructor Initialization
6) STL Redemption

References

[1] Arxiv.org. 2020. [online] Available at: <https://arxiv.org/pdf/1910.01321> [Accessed 25 November 2020].

[2]2020. [Online]. Available: https://www.researchgate.net/publication/303954569_Static_Analysis_of_Security_Vulnerabilities_in_CC_Applications. [Accessed: 25- Nov- 2020].

[3]J. Boyd, "Reviewing C/C++ Code for Security Vulnerabilities", *Blog.securityinnovation.com*, 2020. [Online]. Available: https://blog.securityinnovation.com/blog/2014/06/reviewing-cc-code-for-security-vulnerabilities.html. [Accessed: 25- Nov- 2020].

[4]"sinh - C++ Reference", *Cplusplus.com*, 2020. [Online]. Available: https://cplusplus.com/reference/cmath/sinh/. [Accessed: 25- Nov- 2020].

[5]"C++ Program to Illustrate Trigonometric functions - GeeksforGeeks", *GeeksforGeeks*, 2020. [Online]. Available: https://www.geeksforgeeks.org/c-program-to-illustrate-trigonometric-functions/. [Accessed: 25- Nov- 2020].

[6]*Index-of.es*, 2020. [Online]. Available: http://index-of.es/Miscellanous/24-DEADLY-SINS-OF-SOFTWARE-SECURITY-2010.pdf. [Accessed: 27- Nov- 2020].