Abertay University

Buffer Overflows: The What and How

Isaac Basque-Rice – 1901124@uad.ac.uk

Introduction to Security – CMP110

BSc Ethical Hacking Year 1

2019/20

Note that Information contained in this document is for educational purposes.

Abstract

Buffer overflows are, in short, when the buffer in a program (the space in memory where data that is being transferred from one location to another is stored) is being asked to hold more data than it is strictly speaking allowed or able to. The results of this problem are varied and have the possibility to be quite harmful in numerous ways

To this end the aim of this paper is to explain, give examples of, and demonstrate what buffer overflows are, with an emphasis on how they relate to security, specifically regarding the ability to remotely execute code on target machines vulnerable to buffer overflow exploitation.

The tasks that were undertaken as a part of this paper were as follows:

Task 1 was developing a simple C++ program that was vulnerable to buffer overflow exploitation, the purpose of this was to show how easy it is to implement a mistake such as this through the creation of an app that could, feasibly, be one a beginner may create.

Task 2, on the other hand was exploiting a previously existing, vulnerable program, in this case Destiny Media Player, a Windows XP media player program that can be overflown with an overly long playlist file. The purpose of this task is to serve as an example of a real world vulnerable program, and show how one may go about exploiting this vulnerability.

The result of these tasks was to demonstrate that the possibility for buffer overflows to be exploited is very easy to implement, and has been on numerous occasions, specifically with regards to Destiny Media Player.

CONTENTS

1 INTRODUCTION

1.1 BACKGROUND

The purpose of this paper is to educate the reader on the concept of a Buffer Overflow and to demonstrate an attack using this method to an extent that it can be replicated by a peer. To this end, it is important that some background information is given around Buffer Overflows, exactly what they are, historic implementations and vulnerabilities, and ultimately, touch on protections against this kind of attack.

1.1.1 What is a buffer overflow?

To answer the question of what a buffer overflow is, it's necessary to clarify the meaning of a "buffer" as it relates to computing. A buffer is, in short, a storage location in memory that persists for a relatively short period of time. Probably the most well-known implementation of a buffer is in online video and audio files such as those hosted on YouTube and Spotify etc. The process of buffering (loading data into the buffer) is well known to anyone who has ever had a particularly slow internet connection. (Christensson, 2006)

Buffering is not, however, reserved only for files delivered across networks. Most programs load necessary data into the buffer mid execution in order to avoid stuttering when said data is transferred from one location to another, typically during input/output operations.

Mistakes that the developer of a program makes (particularly in more "low-level" programming languages such as C and C++ that allow for finer memory management on the developer's part) can allow for more data than the program can handle to be written into the buffer (Veracode, n.d.). The best-case scenario in this instance is a localised program crash, a significantly worse possibility is a full system crash, however, provided the system isn't damaged by the crash and lost data can be recovered, this isn't as catastrophic an event as one may imagine. The worst-case scenario, however, and a likely one at that, is that a buffer overflow is a deliberate act performed by a malicious actor to exploit the system on which the attack was launched.

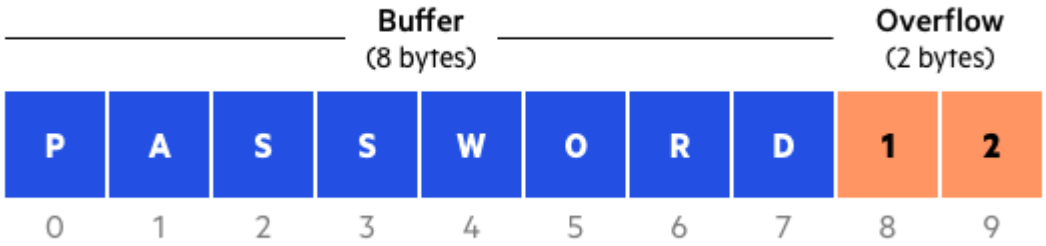


Figure 1, an example of a buffer overflow, where the buffer is an 8-byte string and data that isn't supposed to be accessed (in this case the number 12) is accessed through an overflow (Imperva, n.d.)

1.1.2 How can threat actors make use of this vulnerability?

With adequate knowledge of the system or application the actor is exploiting (i.e. the memory layout of the system, how much data is allowed in buffers etc.) they can inject malicious code into the system by overwriting a pointer declared after the buffer that may have been used for a legitimate reason, and redirecting it to the malicious code's memory location and executing it. This is a generic example of arbitrary code execution through a buffer overflow exploit (Imperva, n.d.).

A process that is essentially identical to the Buffer Overflow attack was identified by a US Government Report in the early 70s, wherein the author stated "the major weaknesses of contemporary operating systems occurs at the interface between the system and the user […] By supplying addresses outside of the space allocated to the user's program, it is often possible to get the monitor to obtain unauthorized (sic.) data for that user"

The first notable exploitation of this vulnerability, however, (and indeed the first notable computer worm) was the Morris Worm, created by R.T. Morris and distributed to the MIT Campus in November of 1988. The worm exploited a buffer overflow vulnerability in the UNIX `finger` protocol which allowed users to find the information of any other user on a network (Harley, 2008), it was mainly intended in order to demonstrate the fairly lax approach to cybersecurity MIT displayed at the time, however it quickly grew out of hand and had a Denial of Service type effect on the network, resulting in Morris becoming one of the first people prosecuted for computer crime in the US (and the world) (Federal Bureau of Investigation, 2018).

Figure 2, Robert Tappan Morris (left), the author of the Morris Worm, alongside a floppy disk containing the source code for the worm (Today In Infosec, 2017)

Another notable exploit comes in the form of the SQL Slammer worm (vulnerability: CVE-2002-0649) which exploited MS SQL Server 2000 and allowed for arbitrary code execution through a buffer overflow attack. The attack involved sending single-byte values to an MS SQL server which the server would then try to parse as legitimate information. The issue arose when the function that returned the required value didn't expect a NULL value to be returned and, as such, caused a crash. The author of the initial exploit then sent an "overly long hostname" to the server and discovered the exploit (Litchfield, 2010).

Finally, the infamous EternalBlue vulnerability developed by the United States' National Security Agency and revealed by the hacker group the Shadow Brokers in April 2017 makes use of three individual bugs in the Windows Server Message Block (SMB) Protocol, of which one makes significant use of a Buffer Overflow vulnerability. The "Wrong Casting Bug", or Bug A in the three-part exploit, use an issue in the processing functions at the kernel level of File Extended Attributes (FEAs, a small amount of metadata within files), specifically the

conversion of an Os2 FEA List to an NT FEA List (Windows' legacy and modern system kernels, respectively), which allows, eventually, for Remote Code Execution on a machine (Grossman, 2017) (Sanchez, 2017).

Figure 3, an image of the WannaCry ransomware. EternalBlue was reportedly utilized by the North Korean government to create this piece of malware that temporarily shut down the NHS's IT systems (Hartley-Parkinson, 2017)

1.1.3 Summary

In summary, a Buffer Overflow is a varied and complex, yet fundamentally extremely simple attack vector that can be and has been used in a diverse set of ways. It is of extreme importance that a security professional is well versed in this form of attack and the aim of this project is to form a baseline on which said professional may improve their knowledge of this area.

To this end what follows is an outline of the aims of the project insofar as what the objective of the work is, the process by which the work will be carried out, and ultimately, the intended result of the project.

1.2 AIM

The intentions of this project are, as previously mentioned, to educate the reader on the nature of buffer overflows to the extent that they could hypothetically replicate the steps outlined in this paper in order to exploit a vulnerable system. The methods by which the paper will attempt to achieve this are twofold, firstly, to build a very small C++ program in order to further explore the nature of buffer overflows on the developer's side (i.e. the mistakes one can make and how to mitigate them), and then subsequently exploit an existing application with a known vulnerability.

2 PROCEDURE

2.1 OVERVIEW OF PROCEDURE

As mentioned previously, the procedure this paper will follow has two primary steps, the creation of a simple, vulnerable program in order to demonstrate the concept of the buffer overflow further, and then an exploit "in the wild", so to speak, wherein a program already determined to be vulnerable will be exploited.

The intent of the paper with regards to the latter is to exploit a local buffer overflow vulnerability found in version 1.61.0 of Destiny Media Player, a media player built for Windows XP that "helps users listen to their favorite online radio stations and play the audio content from files with the following formats: MP3, MPE, WAV, MIDI, ASF, WMA, AVI and MPEG, as well as from audio CDs.". The exploit in particular is described in CVE-2009-3429 and allows users to exploit shellcode using an overly long ".lst" (playlist) file, essentially allowing the user to exploit the system by adding too many files/songs to a playlist.

2.2 PROCEDURE PART 1 – A SIMPLE PROGRAM

Writing a simple program to demonstrate buffer overflows, the source code for these programs can be found in Appendix A, Screenshots of runtime can be found in appendix B.

The code found in AppendixA, CodeA Is a simple program designed to take two numbers, add them together, and output the result, however the introduction of a buffer array variable (buffer[5]), in tandem with a for loop which inputs too many values into the array than it can theoretically handle, results in whatever the user inputs being overridden and the output being the same.

2.3 PROCEDURE PART 2 – DESTINY MEDIA PLAYER

The process of attacking Destiny Media Player in this way is significantly more drawn out than the previous procedure, this is self-evident to an extent in that it's an application that was not created for the purpose of this paper, and as such will take longer to exploit, however this is not the only reason the process will take longer.

Destiny Media Player has been out of development for some years now, and as such there is no support for it on any modern operating system, the initial plan of this paper was to make use of the pre-prepared labs the university offers, however due to the COVID-19 pandemic this became impossible. This means a Windows XP Virtual Machine must be set up for the purposes of this paper. Acquiring an ISO and a copy of Destiny, setting it up, and subsequently creating an environment in which the application can be exploited adequately is, have no doubt, a relatively lengthy process.

This process included sourcing a Windows XP ISO from the Internet Archive found here (Microsoft, 2002), and a copy of Destiny Media Player 1.61.0 here (Marculescu, 2013).

Unfortunately, due to the constraints of an out-of-support operating system (Namely Microsoft's lack of support for IE 6.0, seen in Screenshot B, which was the most up-to-date browser on XP SP1), a shared folder between the guest and host operating system had to be set up through which the required exe could be downloaded onto the host machine and passed through. The success of this can be seen in Screenshot C.

After achieving this the intent is to write a Perl script that causes an overload in the way Destiny handles playlist files, thereby crashing the application. Once performing similar steps to installing Destiny Media Player , i.e. finding a deprecated version of strawberry perl to install onto the VM, a script was written to create an .m3u file that would ideally overload the player. This can be found in Screenshot D. The code for this can also be found in Code B.

Upon achieving this goal, the intent is to use a debugger in order to further examine the state the application/machine was in at the time of the crash (i.e. what the memory stack looked like at this time) in order to form a roadmap towards the ultimate goal to exploit the machine, perhaps by strategically placing a pointer in the stack at the correct memory location.

3 RESULTS

3.1 RESULTS FOR PART 1

As shown in Appendix B, Screenshot A, the expected result was achieved, the numbers 3 and 5 were input into the program with the expectation that the result would, of course, be 8, however due to the introduction of the overflown buffer mid-execution, the input values became corrupted.

An important thing to note here is that the values for the variables opA and opB will always be corrupted in the same way, this is due to the fact that the process is not, as one may expect initially, a function of the input values, but instead is totally independent of said values, and are actually corrupted by the function that is passed into the buffer[i] variable during the for loop (in this instance, the value at position x will be equal to x*2)

3.2 RESULTS FOR PART 2

Firstly, in order to test the script functions as properly intended, the code was modified (Code B) to output to a simple .txt file first, Screenshot E demonstrates that this did in fact work as intended, however when modifying said code to instead output a .lst or .m3a (or other filetypes that could be handled by a media player) all this managed to achieve was to launch the player.

After adjusting some values the resulting .lst file generated was 10kb in size, and clicking this file resulted in the player being opened and the user being greeted with a popup saying "Destiny Media Player has encountered a problem and needs to close" (Seen in Screenshot F), this can be interpreted as a successful crash caused by a buffer overflow exploit.

The crash log you find in Crash Log 1 is the contents of the file located at `C:\DOCUME~1\Owner\LOCALS~1\Temp\WER4.tmp.dir00\appcompat.txt` after the crash, this simply contains information about the relevant files. The full crash report, upon further inspection, contains the memory locations where the payload was placed (specifically in this case between 0x00313434 and 0x00315b44 (Screenshot G)), the log further specifies that

the memory address for the exception in question is 0x00000000004266d7, which may prove invaluable in terms of remote code execution.

4 DISCUSSION

4.1 GENERAL DISCUSSION

Up to this point this paper has touched upon the nature of Buffer Overflow attacks and how they both have been, and are being used in a real-world context, with two demonstrations to show this.

The results of the procedures carried out in this paper show two distinct, yet related things. Part 1 demonstrates that buffer overflow vulnerabilities are inherent to the combination of a low level, memory-management-heavy language, and a careless developer, and as a result the utmost caution should be taken when working with these tools. Part 2, on the other hand, demonstrates the real world consequences of part 1 with regards to the results of a careless dev and a widely-released product, and allows for the reader to understand that this exploit is not mainly found within the confines of textbooks and theory, but can be shown today.

The aims of this project have, it seems, been met, in the most superficial of terms, however there is significant room for improvement as far as expansion is concerned. This is to say that objectives could have been “tacked on”, so to speak, without any changes in terms of scope or goal. The most notable of these additions would be a continuation of part 2, namely making use of a debugger to expand on error logs provided internally, and the subsequent continuation of exploitation to ultimately gain full RCE over the machine, which, it appears, is possible.

What follows is methods by which to counter Buffer Overflow attacks from the perspective of a developer, the conclusion to this paper, and finally any future work one may wish to work on relating to buffer overflows.

4.2 COUNTERMEASURES

The foremost and simplest step to take to prevent Buffer Overflow attacks is by not using C or C++ at all in development, as, whilst they are extremely powerful languages, with great

power comes great responsibility, responsibility which, understandably, many developers are unable to handle when it comes to relatively low-level memory management.

However, it is likely that changing the language of an entire codebase is not feasible if it's already written in C/C++, in this instance, there are several solutions that one can implement. The following is a non-exhaustive list of these solutions, what they are, what they do, and how they assist in mitigation.

Address Space Location Randomisation (ASLR): This method is intended to throw a malicious actor off their step by randomly assigning executable code to different memory locations within a system. Many attackers use knowledge of where programs exist in memory to execute their own malicious code, randomising this instead of allowing it to remain linear makes this form of attack extremely difficult to execute well. (Cloudflare, n.d.)

Data Execution Prevention (DEP): DEP is a system whereby if a program is detected using system memory incorrectly, the user of the program is notified, and the program execution is terminated. For example, if the DEP program detects that an unusual memory location is executing a program (DEP monitors a series of these locations intermittently), it will execute its mitigation process. (McDowell, 2019)

Making use of safer functions: Some functions in C, specifically `strcopy()` and the like, make use of special characters to point to where a copy should end, this is bad practice as the special characters exist within the data itself, which can be edited by users (Du, 2017). Unfortunately, C does not provide a standardised way of making these functions safe, however different Operating Systems provide their own fixes, such as Windows's `strcpy_s()` and OpenBSD's `strlcpy()` (Kerestan, 2017).

4.3 CONCLUSIONS

In conclusion, buffer overflow attacks are a real and present danger within computing and have been for many years. Careless developers can easily allow for them to happen , and many (including the developers of Destiny Media Player) already have. It is imperative for these bugs to be discovered and quashed, and for developers themselves to be educated on the conditions that could create the possibility for a buffer overflow exploit.

4.4 FUTURE WORK

Ideally future work would build upon the results of Part 2 as opposed to part one, this is because further analysis can be undertaken into the Destiny Media Player program, and buffer overflow exploits within the confines of apps “in the wild”, so to speak, as a whole.

Work that can be undertaken specifically in the context of Destiny could include making use of a debugger in order to further investigate the contents of the stack at the point of crash. There is no doubt that this information could be invaluable in order to perform a full exploitation of the app, and possibly gain RCE over the whole machine.

In addition to this, work done in the future could also include further research into the exploits laid out in section 1 (EternalBlue etc) in a similar way to how Destiny could be exploited, namely establishing a virtual environment and running any number of attacks against it in an attempt to gain RCE. The goal of further research into presently known Buffer Overflow exploits would, of course, be to discover exploits in currently unexploited applications. This kind of research would perhaps prove the most challenging, but also the most exciting in terms of outcome and possible ramifications.

Finally, it is possible, of course, to commit a buffer overflow attack remotely by using specially crafted FTP packets to a target machine running vulnerable software, an extension of this paper could feasibly be to attempt to modify a similar attack found here to work “over the wire”, if you will.

5 REFERENCES

Anderson, J. P., 1972. *Computer Security Technology Planning Study,* Bedford, Massachusetts: Deputy for Command and Management Systems HQ Electronic Systems Division (AFSC).

Christensson, P., 2006. *Buffer Definition.* [Online]
Available at: https://techterms.com/definition/buffer
[Accessed 2020 February 12].

Cloudflare, n.d. *What is buffer overflow?.* [Online]
Available at: https://www.cloudflare.com/learning/security/threats/buffer-overflow/
[Accessed 24 February 2020].

Du, W., 2017. Chapter 4 - Buffer Overflow Attack. In: *Computer Security: A Hands-on Approach.* Syracuse, New York: CreateSpace Independent Publishing Platform, pp. 18-19.

Federal Bureau of Investigation, 2018. *The Morris Worm.* [Online]
Available at: https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218
[Accessed 21 February 2020].

Grossman, N., 2017. *EternalBlue – Everything There Is To Know.* [Online]
Available at: https://research.checkpoint.com/2017/eternalblue-everything-know/#bugb
[Accessed 26 February 2020].

Harley, D., 2008. *The Morris Worm: a Malware Prototype.* [Online]
Available at: https://www.welivesecurity.com/2008/11/02/the-morris-worm-a-malware-prototype/
[Accessed 21 February 2020].

Hartley-Parkinson, R., 2017. *US publicly blames North Korea for WannaCry attack that brought down NHS systems.* [Online]
Available at: https://metro.co.uk/2017/12/19/us-publicly-blames-north-korea-wannacry-attack-brought-nhs-systems-7170089/
[Accessed 26 February 2020].

Imperva, n.d. *Buffer Overflow Attack.* [Online]
Available at: https://www.imperva.com/learn/application-security/buffer-overflow
[Accessed 18 February 2020].

Kerestan, B., 2017. *How to Detect, Prevent, and Mitigate Buffer Overflow Attacks.* [Online]
Available at: https://dzone.com/articles/how-to-detect-prevent-and-mitigate-buffer-overflow
[Accessed 24 February 2020].

Litchfield, D., 2010. *The Inside Story of SQL Slammer.* [Online]
Available at: https://threatpost.com/inside-story-sql-slammer-102010/74589/
[Accessed 21 February 2020].

Marculescu, A., 2013. *Destiny Media Player.* [Online]
Available at: https://www.softpedia.com/get/Internet/Internet-Radio-TV-Player/Destiny-Media-Player.shtml
[Accessed 8 April 2020].

McDowell, G., 2019. *Configure or Turn Off DEP (Data Execution Prevention) in Windows.* [Online]
Available at: https://www.online-tech-tips.com/windows-xp/disable-turn-off-dep-windows/
[Accessed 24 February 2020].

Microsoft, 2002. *Windows XP Home Edition Service Pack 1 + SP2.* [Online]
Available at: https://archive.org/details/WXPHOMESP1SP2_2002
[Accessed 8 April 2020].

Sanchez, W. G., 2017. *MS17-010: EternalBlue's Large Non-Paged Pool Overflow in SRV Driver.* [Online]
Available at: https://blog.trendmicro.com/trendlabs-security-intelligence/ms17-010-eternalblue/
[Accessed 26 February 2020].

Today In Infosec, 2017. *twitter.* [Online]
Available at: https://twitter.com/todayininfosec/status/926128404054777856
[Accessed 26 February 2020].

Veracode, n.d. *WHAT IS A BUFFER OVERFLOW? LEARN ABOUT BUFFER OVERRUN VULNERABILITIES, EXPLOITS & ATTACKS.* [Online]
Available at: https://www.veracode.com/security/buffer-overflow
[Accessed 18 February 2020].

6 APPENDICES

1. APPENDIX A - CODE

Code A

```
 //A program to demonstrate buffer overflow

#include <iostream>
using namespace std;

int main()
{
    //two operands
    int opA = 0;
    int opB = 0;

    int answer = 0;

    int buffer[5];

    cout << "What numbers would you like to add?" << endl;
    cin >> opA; cin >> opB;

    //sets values in the buffer to far exceed what it is supposed to be able t
o handle
    for (size_t i = 0; i < 20; i++)
    {
        buffer[i] = i * 2; // IDE gives warning on this line
    }

    answer = opA + opB;

    // will not return the two numbers you asked for
    cout << "The result of adding " << opA << " and " << opB << " is " << answ
er;
}
```

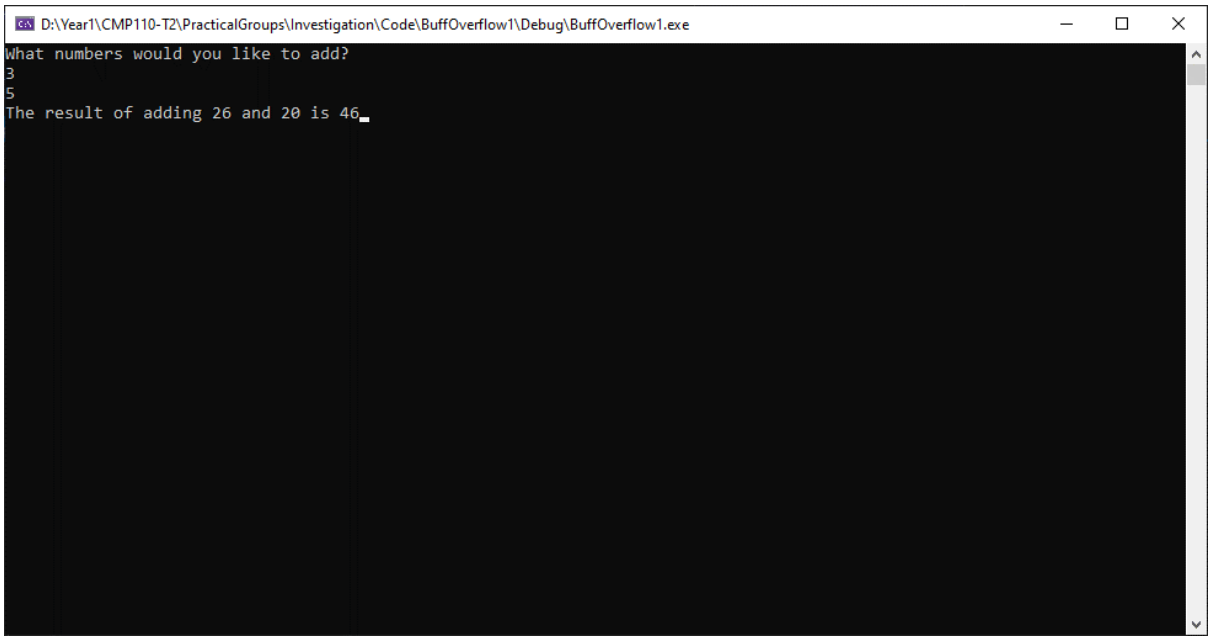
Code B

```
#My Perl program to crash DMP

my $file= "destinyCrash.txt";

my $junk1 = "/x41" x10000;
```

```
my $eip = pack('V', 0x00313438);

open($FILE, ">$file");
print $FILE $junk1;
close($FILE);
```

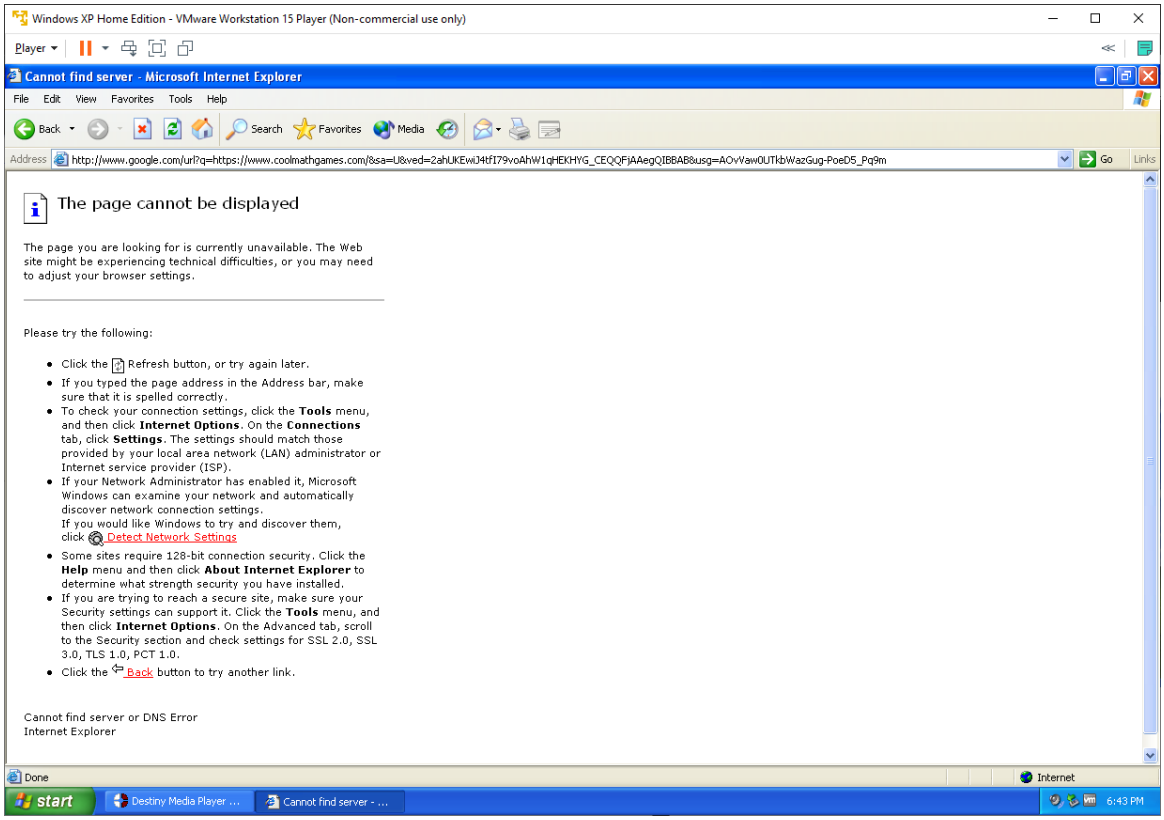
1. APPENDIX B – SCREENSHOTS

Screenshot A



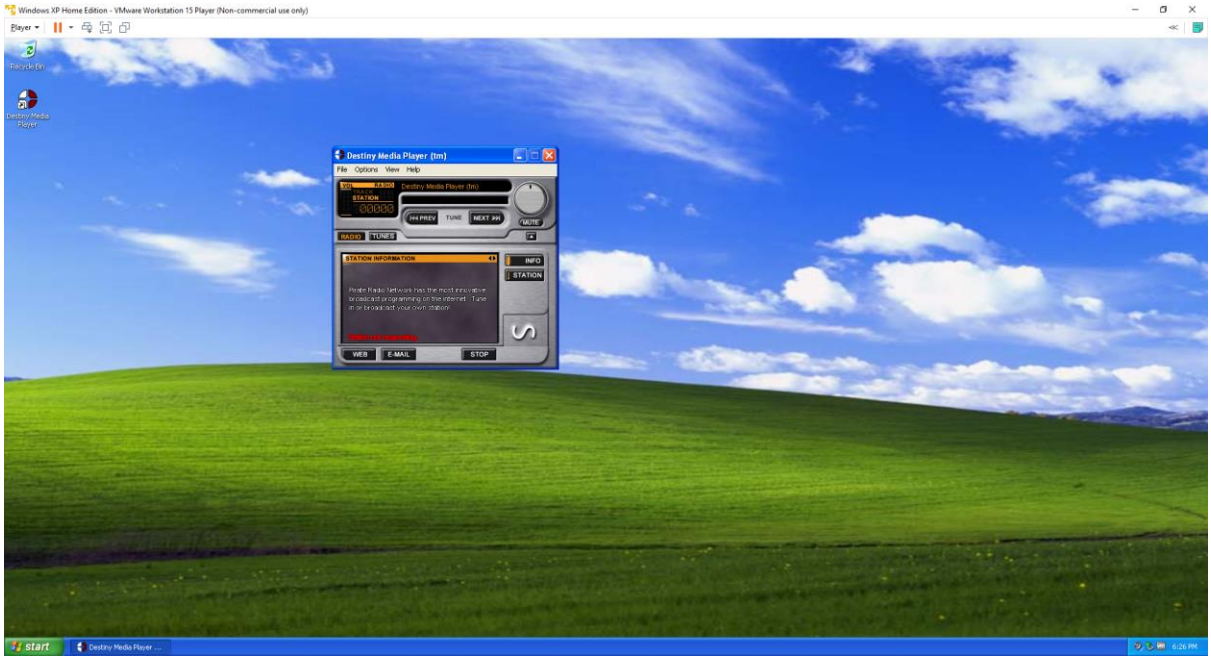
Screenshot 1, Code A during runtime where the numbers input are 3 and 5

Screenshot B



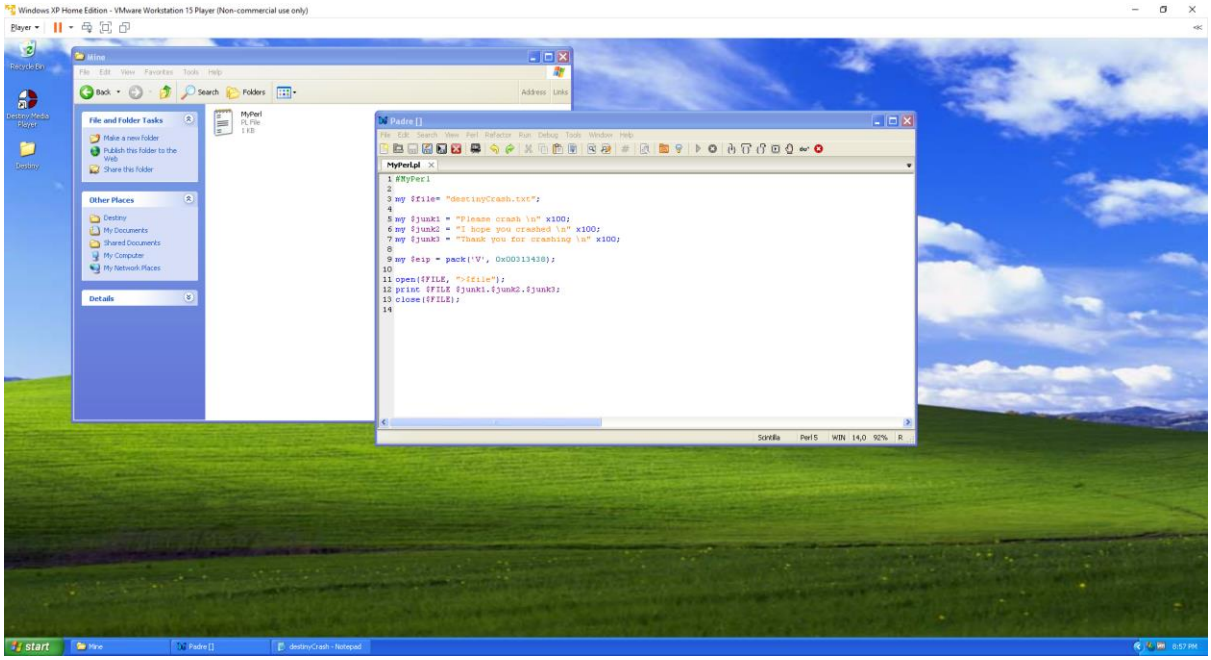
Screenshot 2, IE 6.0 failure to connect message, found on the vast majority of websites

Screenshot C



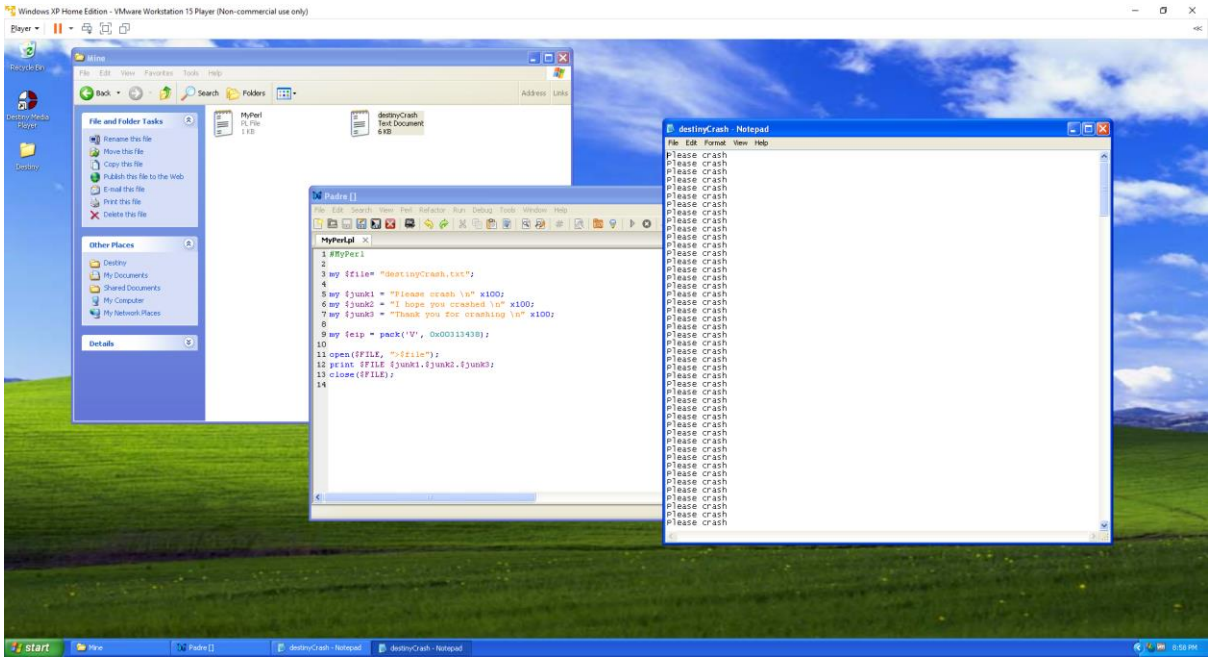
Screenshot 3, Windows XP inside a Virtual Machine running Destiny Media Player

Screenshot D



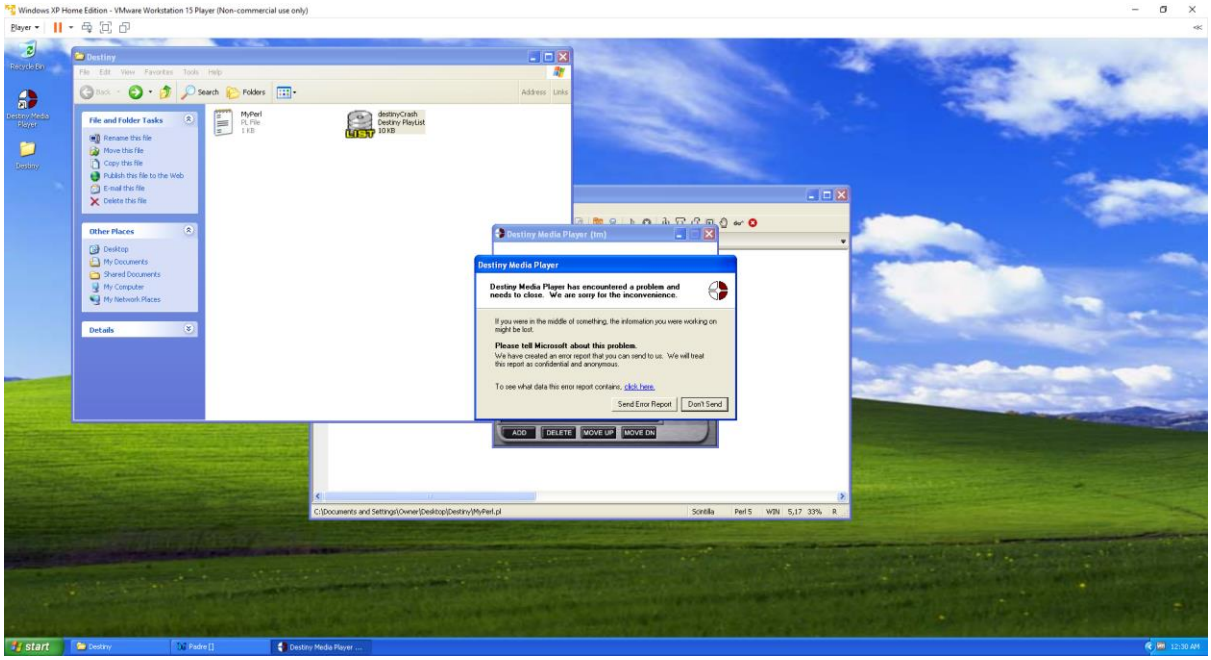
Screenshot 4, an attempt at a perl script to overload Destiny's playlist handler and cause it to crash

Screenshot E



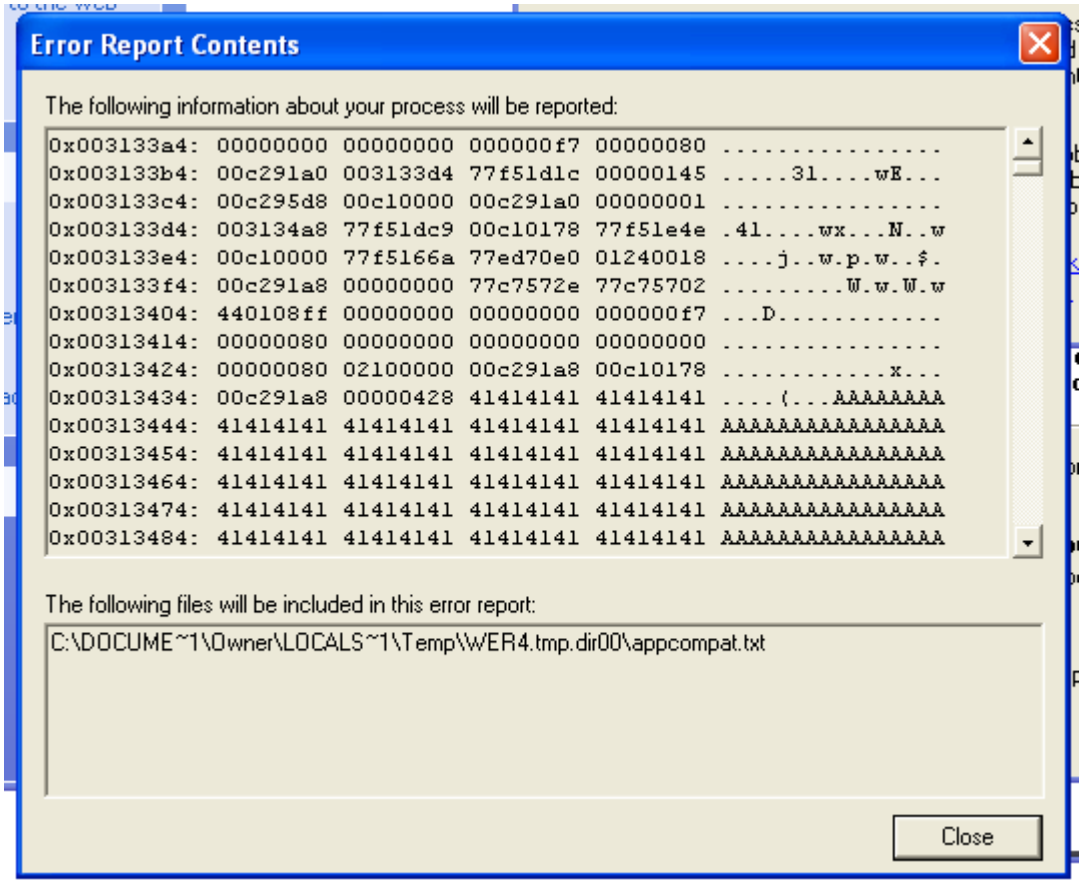
Screenshot 5, Showing the attempt worked properly

Screenshot F



Screenshot 6, Crash file being opened

Screenshot G



Screenshot 7, Error Report Contents

2. APPENDIX C - LOGS

Crash Log 1

<?xml version="1.0" encoding="UTF-16"?>

<DATABASE>

<EXE NAME="Destiny.exe" FILTER="GRABMI_FILTER_PRIVACY">

<MATCHING_FILE NAME="Destiny.exe" SIZE="1437696" CHECKSUM="0xEA3FCB15" BIN_FILE_VERSION="1.3.1.0" BIN_PRODUCT_VERSION="1.3.1.0" PRODUCT_VERSION="1, 3, 1, 0" FILE_DESCRIPTION="Destiny Media Player" COMPANY_NAME="Destiny Software" PRODUCT_NAME="Destiny Media Player" FILE_VERSION="1, 4, 3, 0" ORIGINAL_FILENAME="Destiny.exe" INTERNAL_NAME="DMP" LEGAL_COPYRIGHT="Copyright © 2002" VERFILEDATEHI="0x0" VERFILEDATELO="0x0" VERFILEOS="0x40004" VERFILETYPE="0x1" MODULE_TYPE="WIN32" PE_CHECKSUM="0x0" LINKER_VERSION="0x0" UPTO_BIN_FILE_VERSION="1.3.1.0" UPTO_BIN_PRODUCT_VERSION="1.3.1.0" LINK_DATE="01/14/2003 19:49:41" UPTO_LINK_DATE="01/14/2003 19:49:41" VER_LANGUAGE="English (United States) [0x409]" />

<MATCHING_FILE NAME="uninst.exe" SIZE="44247" CHECKSUM="0x1318FA05" MODULE_TYPE="WIN32" PE_CHECKSUM="0x0" LINKER_VERSION="0x0" LINK_DATE="02/07/2004 17:26:28" UPTO_LINK_DATE="02/07/2004 17:26:28" />

</EXE>

<EXE NAME="kernel32.dll" FILTER="GRABMI_FILTER_THISFILEONLY">

<MATCHING_FILE NAME="kernel32.dll" SIZE="930304" CHECKSUM="0xCBCCF8A9" BIN_FILE_VERSION="5.1.2600.1106" BIN_PRODUCT_VERSION="5.1.2600.1106" PRODUCT_VERSION="5.1.2600.1106" FILE_DESCRIPTION="Windows NT BASE API Client DLL" COMPANY_NAME="Microsoft Corporation" PRODUCT_NAME="Microsoft® Windows® Operating System" FILE_VERSION="5.1.2600.1106 (xpsp1.020828-1920)" ORIGINAL_FILENAME="kernel32" INTERNAL_NAME="kernel32" LEGAL_COPYRIGHT="© Microsoft Corporation. All rights reserved." VERFILEDATEHI="0x0" VERFILEDATELO="0x0"

VERFILEOS="0x40004" VERFILETYPE="0x2" MODULE_TYPE="WIN32"
PE_CHECKSUM="0xE7ED3" LINKER_VERSION="0x50001"
UPTO_BIN_FILE_VERSION="5.1.2600.1106"
UPTO_BIN_PRODUCT_VERSION="5.1.2600.1106" LINK_DATE="08/29/2002 10:40:40"
UPTO_LINK_DATE="08/29/2002 10:40:40" VER_LANGUAGE="English (United States)
[0x409]" />

</EXE>

</DATABASE>