# Flare-On 5: Challenge 10 Solution – golf.exe

**Challenge Author: Ryan Warns**

## Summary

This challenge leverages the Intel VT-x instruction set to create a very thin hypervisor which interacts with the usermode binary to implement the algorithm for the challenge. The VT-x (and analogous AMD-V) are Virtual Machine Extensions VMX), instruction sets that provide a framework that developers can use to monitor hardware access, allowing a straightforward way to implement a hypervisor. Although not seen in the wild, the idea of using this instruction set in malware was first published at IEEE Oakland in May 2006[1] and at BlackHat 2006[2]. This challenge implements a Type 2 hypervisor which is a hypervisor implemented as a driver running on a host operating system. Type 1 hypervisors run directly on hardware. As we will see in this challenge, the complexity of the instruction set and large amount of indirection require background knowledge in order to pinpoint what code to pay attention to.

## Triage

CFF explorer reveals that `golf.exe` is a 64-bit Windows executable. `strings.exe` reveals several interesting strings.

```
SYSTEM\CurrentControlSet\Control
SystemStartOptions
TESTSIGNING
ZwLoadDriver
ntdll
SeLoadDriverPrivilege
SYSTEM\CurrentControlSet\services\fhv
ErrorControl
Start
Type
\??\%s\fhv.sys
ImagePath
\Registry\Machine\System\CurrentControlSet\Services\fhv
C:\fhv.sys
ZwUnloadDriver
Too bad so saddd %x
%s@flare-on.com
RSDS|78.VR
```

---

[1] https://ieeexplore.ieee.org/document/1624022

[2] https://www.blackhat.com/html/bh-usa-06/bh-usa-06-speakers.html#Rutkowska

```
t:\objchk_win7_amd64\amd64\golf.pdb
```

**Figure 1 - Strings found in golf.exe**

Several of these strings already give us a clue as to what the binary might do:

- `ZwLoadDriver`[3] is the internal Windows API call that loads a driver specified by the registry
    - Note the string `SeLoadDriverPrivilege` as well
- The `\Registry\Machine\...` string is the registry key corresponding to Sservices on the system
    - `\Registy\Machine\` is the naming schemed used in registry paths by kernel code and corresponds to HKLM
    - The `CurrentControlSet\Services` registry hive lists all installed services on the system. This key is usually automatically created and populated when you run sc start
- Multiple references to fhv.sys might point us at a driver file that the binary uses
    - .sys files are PE files that are meant to be loaded and run in kernel/ring-0 space.

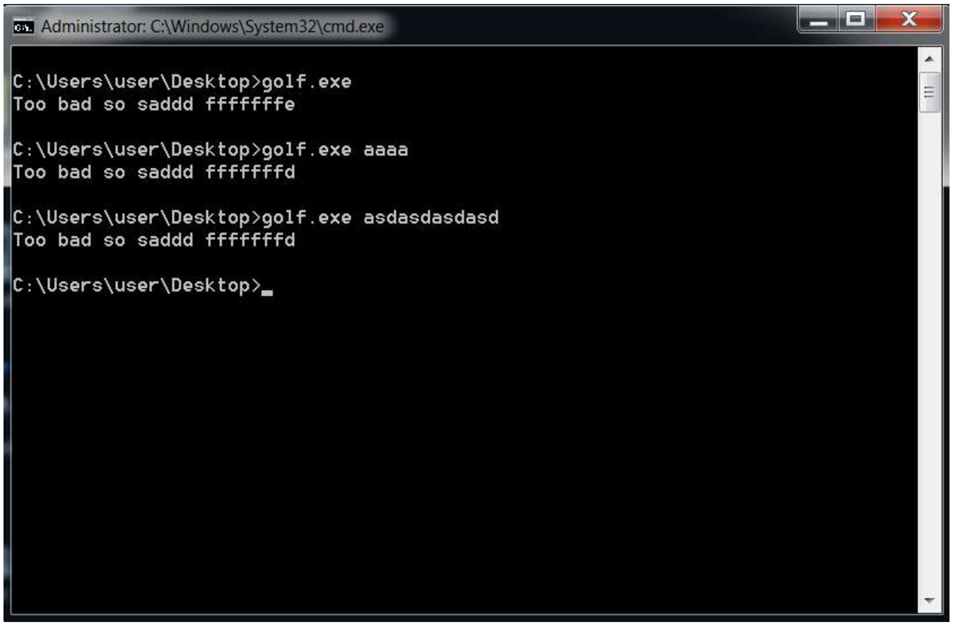
Using these strings as a starting point we can open `golf.exe` in IDA. Since we already have a hunch that the binary may be starting a binary we can use cross references to those strings as a place to start looking. The driver-related strings from Figure 1 are all referenced in the subroutine at address `0x100001700`. Looking at the API usage in this function we can see the following:

- Dynamically resolving `ZwLoadDriver`
- Making several Registry-related API calls, likely setting up the services key described earlier
- At address `100001A0B` we can see a call instruction using the `ZwLoadDriver` function pointer
- Deleting a file named `fhv.sys`

Following the cross references to `sub_0x100001700` up to the main entry point we can see there are several branches that the code can take before running this subroutine. If we run `procmon` we won't see any events related to the registry keys or driver file name. If we assume or guess that `golf.exe` accepts the key on the command line we will notice different error messages

---

[3] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-zwloaddriver



```
C:\Users\user\Desktop>golf.exe
Too bad so saddd fffffffe

C:\Users\user\Desktop>golf.exe aaaa
Too bad so saddd fffffffd

C:\Users\user\Desktop>golf.exe asdasdasdasd
Too bad so saddd fffffffd

C:\Users\user\Desktop>_
```

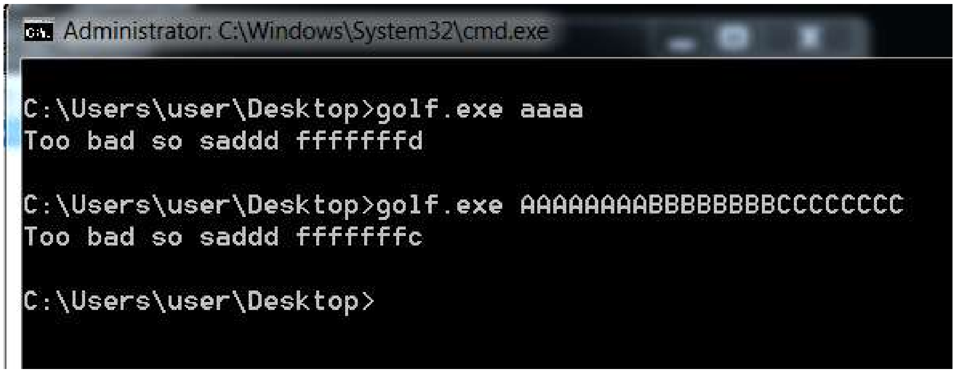
**Figure 2 - Output when running golf.exe with command line parameters**

At this point we are unlikely to get any more information from basic analysis and should start debugging the binary.

## Initial Analysis – golf.exe

Since we know just running the executable doesn't reach the function we believe loads the driver it's usually a good idea to step out to the main function to see where our sample is diverging from what we want; there might be an environment or other check happening that's causing the binary to run different functionality.

The main function is at address `0x100001C10`. The first branch checks to see if `argc` is greater than two. This confirms our guess above that the sample is accepting input on the command line. The second branch happens in response to a `strlen` call on the second command line argument. If the result of `strlen` is anything other than 24 the binary exits. We can confirm this on the command line:



```
C:\Users\user\Desktop>golf.exe aaaa
Too bad so saddd fffffffd

C:\Users\user\Desktop>golf.exe AAAAAAAABBBBBBBBCCCCCCCC
Too bad so saddd fffffffc

C:\Users\user\Desktop>
```

**Figure 3 - golf.exe run with the correct length input parameter**

We now know the sample accepts a string of length 24. The next subroutine called is at address `0x100001A60`. We see this function using the string `fhv.sys` and a reference to the subroutine at `0x100001700`, our potential driver load function.

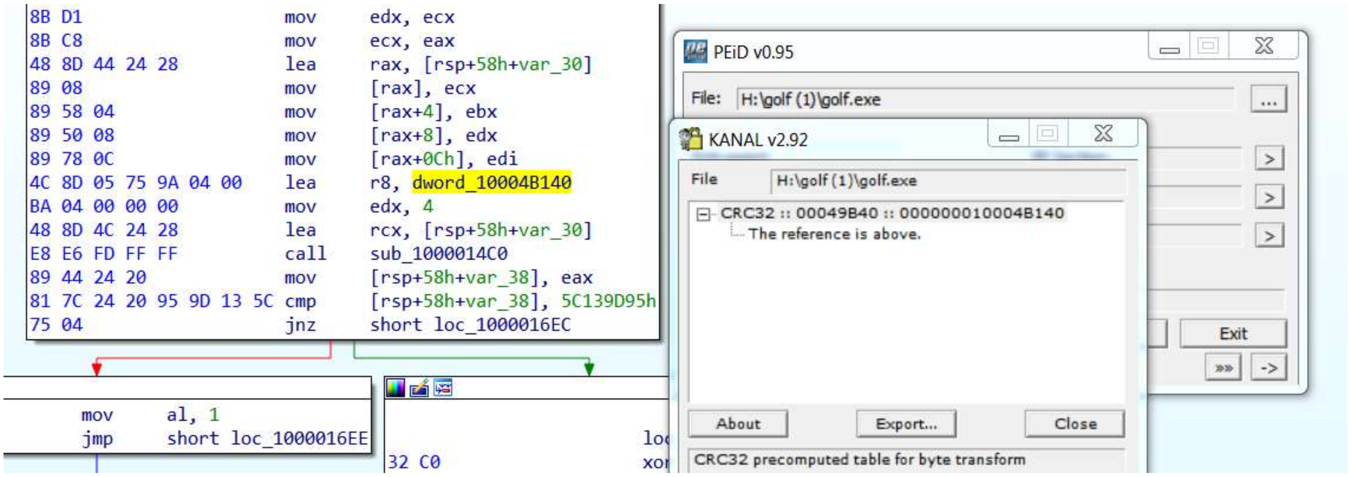
The first branch happens in response to `sub_100001680`. This function is relatively small, executing the `cpuid` instruction before calling `sub_1000014C0` and comparing the result to `0x5C139D95`. The `cpuid` instruction is used to query information about processor and what functionality it supports. The documentation[4] for this instruction says that the information being queried is indicated by the value in the EAX register (also called a `leaf` in some documentation). This function sets eax to `0x40000001` before executing the `cpuid` instruction. Some light googling[5] will reveal that the leaves between `0x40000000` and `0x400000FF` are reserved for hypervisor software use and have no defined meaning to the processor.

The result of this function is passed to `sub_0x1000014C0`. This routine also takes a global buffer at `0x10004B140` and runs an unknown algorithm on the input. In cases like this the first place we should usually consult is `PEiD's KANAL` plugin, or a similar automated cryptography scanner. Doing so will reveal that this algorithm looks like a CRC32 calculation.

---

[4] https://c9x.me/x86/html/file_module_x86_id_45.html

[5] https://lwn.net/Articles/301888/



```
8B D1                mov     edx, ecx
8B C8                mov     ecx, eax
48 8D 44 24 28       lea     rax, [rsp+58h+var_30]
89 08                mov     [rax], ecx
89 58 04             mov     [rax+4], ebx
89 50 08             mov     [rax+8], edx
89 78 0C             mov     [rax+0Ch], edi
4C 8D 05 75 9A 04 00 lea     r8, dword_10004B140
BA 04 00 00 00       mov     edx, 4
48 8D 4C 24 28       lea     rcx, [rsp+58h+var_30]
E8 E6 FD FF FF       call    sub_1000014C0
89 44 24 20          mov     [rsp+58h+var_38], eax
81 7C 24 20 95 9D 13 5C cmp  [rsp+58h+var_38], 5C139D95h
75 04                jnz     short loc_1000016EC

     mov     al, 1
     jmp     short loc_1000016EE

32 C0
```

PEiD v0.95

File: H:\golf (1)\golf.exe

KANAL v2.92

File H:\golf (1)\golf.exe

```
CRC32 :: 00049B40 :: 000000010004B140
  The reference is above.
```

CRC32 precomputed table for byte transform

Figure 4 - CRC32 confirmation on the result of CPUID

The CRC of the result from CPUID `0x40000001` should be `0x5C139D95`. During analysis this is not the case, but luckily for us the binary will continue if the CRC does not match.

The next subroutine only performs a registry query to check the contents of HKLM\\SYSTEM\\CurrentControlSet\\Control\\SystemStartOptions. If this registry key contains the string TESTSIGNINGON then this function returns true.

Per MSDN[6], TESTSIGNING is a boot option that can be set to loosen the signature requirements for kernel code. Normally .sys binaries are required to be signed using a certificate signed or validated by Microsoft. If the .sys file is not signed or the signature is invalid the user will be prompted with an error message and the binary will not load.

We can set the signature policy using the bcdedit command and rebooting:

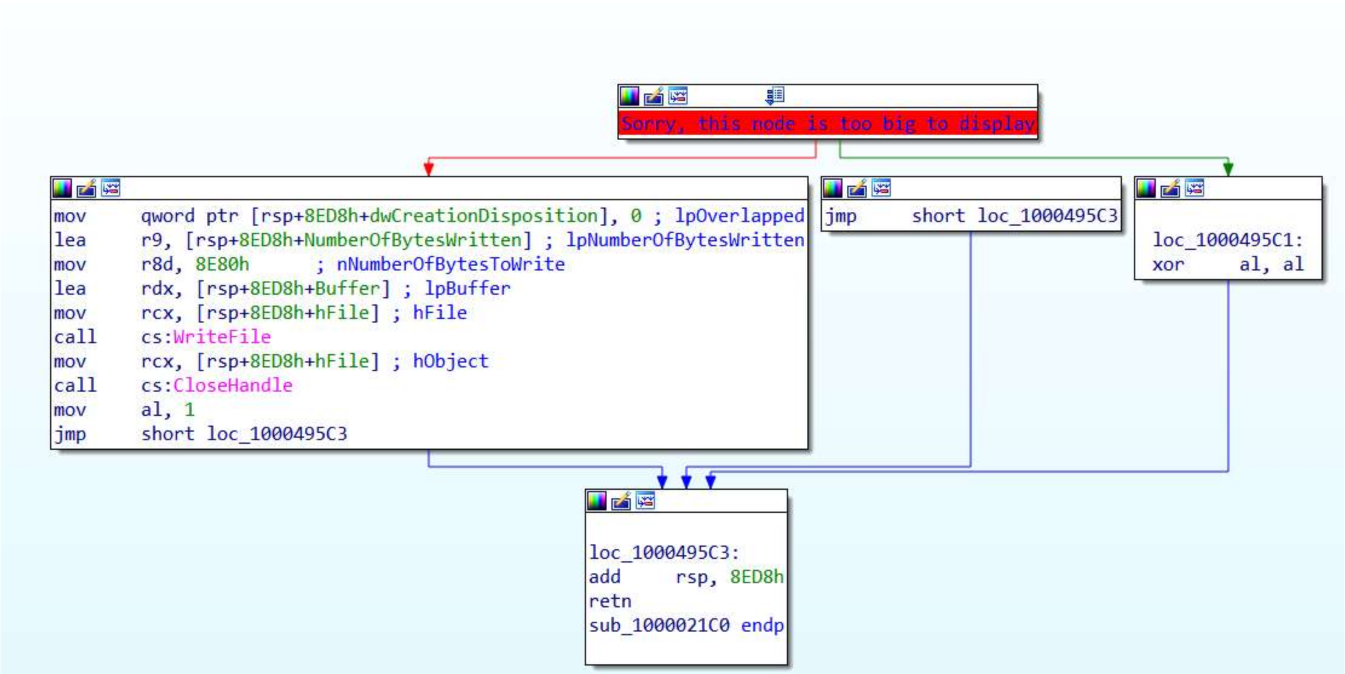
```
bcdedit -set  TESTSIGNING ON
```

Figure 5 - bcdedit command to enable test signing

Unfortunately, running `golf.exe` after enabling test-signing results in the same error shown in Figure 3. The last function to analyze before the driver loading functionality is the subroutine at address `0x1000021C0`. This function is incredibly large and may break IDA's graph functionality. Fortunately, we can still see the `WriteFile` API call in the graph and cross references from this function; we can assume this function drops the .sys file to disk as `C:\fhv.sys` before trying to load it. Because we already saw a call to delete the .sys file from disk, stepping over this function but not running the rest of the binary allows us to pull `fhv.sys` from the system.

---

[6] https://docs.microsoft.com/en-us/windows-hardware/drivers/install/the-testsigning-boot-configuration-option



```
Sorry, this node is too big to display
```

```
mov     qword ptr [rsp+8ED8h+dwCreationDisposition], 0 ; lpOverlapped
lea     r9, [rsp+8ED8h+NumberOfBytesWritten] ; lpNumberOfBytesWritten
mov     r8d, 8E80h      ; nNumberOfBytesToWrite
lea     rdx, [rsp+8ED8h+Buffer] ; lpBuffer
mov     rcx, [rsp+8ED8h+hFile] ; hFile
call    cs:WriteFile
mov     rcx, [rsp+8ED8h+hFile] ; hObject
call    cs:CloseHandle
mov     al, 1
jmp     short loc_1000495C3
```

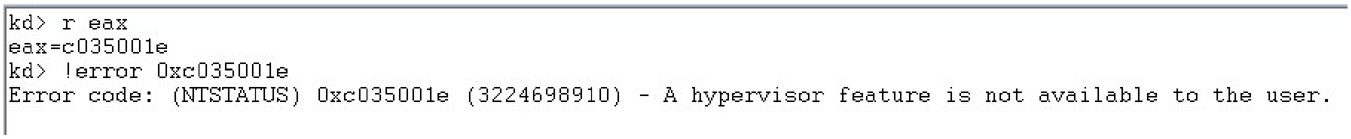
```
jmp     short loc_1000495C3
```

```
loc_1000495C1:
xor     al, al
```

```
loc_1000495C3:
add     rsp, 8ED8h
retn
sub_1000021C0 endp
```

Figure 6 - Function to write fhv.sys to disk

Stepping through our driver-related subroutine at address `0x100001700` allows us to confirm our earlier assumption that this function is manually creating a services registry key before calling `ZwLoadDriver`. We can set a breakpoint directly on this call to learn that the driver is failing to load. If we're using `windbg` we can use the `!error` command to get a description of the return value.



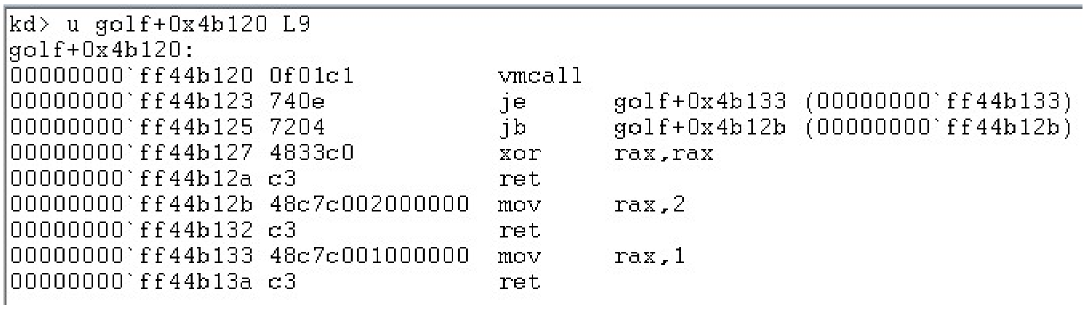
```
kd> r eax
eax=c035001e
kd> !error 0xc035001e
Error code: (NTSTATUS) 0xc035001e (3224698910) - A hypervisor feature is not available to the user.
```

Figure 7 - Error returned from ZwLoadDriver

Based on the error code, namely that the error message doesn't say anything along the lines of "file not found", we can assume that the driver successfully loaded but its entry point failed.

At this point we need to analyze the driver to see why our binary doesn't continue running, but it would be a good idea to finish analyzing `golf.exe` in case it reveals anything we should be on the lookout for when triaging the driver.

Returning the main function, the sample allocates a buffer of RWX data before copying the byte sequence at address `0x10004B120`. This new buffer is used as a parameter to the subroutines at `0x100001E40`, `0x100001F20`, `0x100002000`, and `0x1000020E0`. Because the buffer is allocated as `PAGE_EXECUTE_READWRITE` (0x40) we can disassemble this data as code:

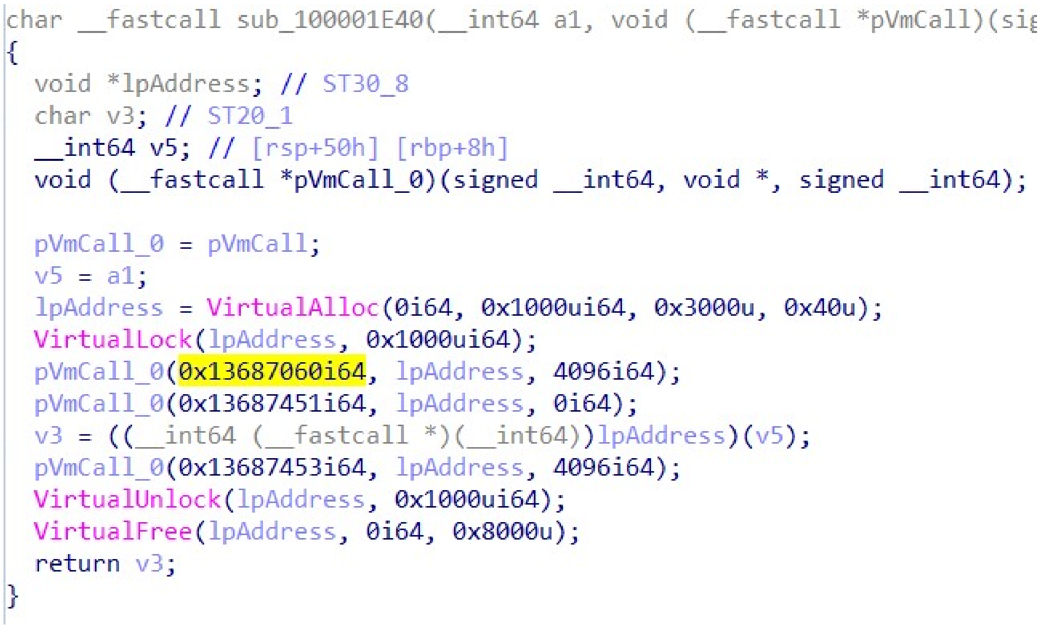


```
kd> u golf+0x4b120 L9
golf+0x4b120:
00000000`ff44b120 0f01c1          vmcall
00000000`ff44b123 740e            je      golf+0x4b133 (00000000`ff44b133)
00000000`ff44b125 7204            jb      golf+0x4b12b (00000000`ff44b12b)
00000000`ff44b127 4833c0          xor     rax,rax
00000000`ff44b12a c3              ret
00000000`ff44b12b 48c7c002000000  mov     rax,2
00000000`ff44b132 c3              ret
00000000`ff44b133 48c7c001000000  mov     rax,1
00000000`ff44b13a c3              ret
```

**Figure 8 - vmcall code copied to a new buffer**

`vmcall` is not a common instruction because it is part of the Intel VT-x instruction set. The documentation for this instruction reveals that this it is used to make an unspecified request to a Virtual Machine Monitor (VMM). We will see later that a VMM is functionally synonymous with a hypervisor. In this scenario this is likely implemented in the `fhv.sys` driver.

The functions that take this vmcall pointer all function similarly: after allocating memory: the function allocates RWX memory, makes several calls using `vmcall` buffer, and executes the newly allocated buffer.



```
char __fastcall sub_100001E40(__int64 a1, void (__fastcall *pVmCall)(sig
{
  void *lpAddress; // ST30_8
  char v3; // ST20_1
  __int64 v5; // [rsp+50h] [rbp+8h]
  void (__fastcall *pVmCall_0)(signed __int64, void *, signed __int64);

  pVmCall_0 = pVmCall;
  v5 = a1;
  lpAddress = VirtualAlloc(0i64, 0x1000ui64, 0x3000u, 0x40u);
  VirtualLock(lpAddress, 0x1000ui64);
  pVmCall_0(0x13687060i64, lpAddress, 4096i64);
  pVmCall_0(0x13687451i64, lpAddress, 0i64);
  v3 = ((__int64 (__fastcall *)(__int64))lpAddress)(v5);
  pVmCall_0(0x13687453i64, lpAddress, 4096i64);
  VirtualUnlock(lpAddress, 0x1000ui64);
  VirtualFree(lpAddress, 0i64, 0x8000u);
  return v3;
}
```

**Figure 9 - Decompilation of sub_100001E40**

Because there is no other functionality in golf.exe, the majority of the logic for this challenge must be in `fhv.sys`, including an implementation of a some form of hypervisor.

# Intel VT-x Instruction Set

## Summary

The Intel VT-x instruction set is a set of instructions that enable to processor to be configured to cause traps (referred to as `VM exits` in the documentation) on certain events, usually related to accessing hardware. These VM exits redirect execution to a hypervisor component which can virtualize and manage this hardware. The full instruction set is documented in Chapter 24 of Volume 3a in the Intel Software Developer's Manual[7] but there are multiple[8] open[9] source[10] implementations[11] of hypervisors of varying complexity to give source code examples.

This section of the manual is meant to serve as a high-level summary of how a Type 2 hypervisor works to give clarity when analyzing `fhv.sys`.

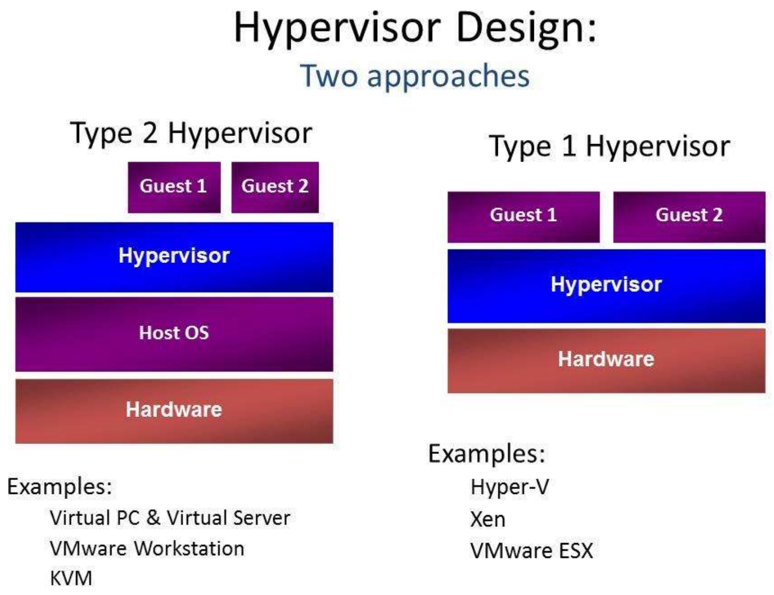


**Figure 10 - Hypervisor diagram from https://blogs.technet.microsoft.com/chenley/2011/02/09/hypervisors/**

The purpose of the VT-x instruction set is to provide a (relatively) easy-to-use framework to instruct the processor what resources are being virtualized. As an example of a normal use case of where this is needed let's review how virtual memory works:

---

[7] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[8] https://github.com/Bareflank/hypervisor

[9] https://github.com/ionescu007/SimpleVisor

[10] https://github.com/airbus-seclab/ramooflax

[11] https://github.com/tandasat/HyperPlatform

- Virtual Memory is a set of structures managed by the Operating System and CPU that maps virtual addresses, the addresses generally used while programming, to physical addresses resident in RAM
- Each process has its own memory map for virtual addresses, so address `0x200000` in calc.exe is different than address `0x200000` in notepad.exe
- When notepad.exe tries to copy data to virtual address `0x200000` its page table structures dictate that in RAM this buffer is at physical address `0xd0000`
- When calc.exe is executing, it has its own page table structures managed by the control register `cr3`. When calc.exe context switches and begins executing on the processor, the value in `cr3` is swapped out from the notepad structures to the calc structures
- Any physical memory used by notepad can be `paged out` (flushed to disk) while other processes are executing
- Virtual address `0x200000` for calc.exe maps to physical address `0xea0000`

Now consider this scenario with two Virtual Machines running under VMWare Workstation. Basic security principles dictate that if two virtual machines are running on the system they should be unaware of each other (minus through networking or other normal system functionality). Each operating system is going to perform the same Virtual-To-Physical address translation described above. What happens if both operating systems think that they have data at physical address `0xd0000`? There's only one set of RAM on the system and neither VM would function correctly if the other could overwrite its data.

This is where the VT-x instruction set comes in. A hypervisor can be instructed to manage access to physical memory and keep a master record of memory maps of both virtual machines. Effectively this means that when VM 1 accesses what it thinks is physical address 0xd0000 the hypervisor can silently redirect this operation to a virtual address it has allocated for that VM, and likewise for VM 2.

On modern processors this process is probably implemented via Second Layer Address Translation (SLAT) which we will explore later in this manual.

## Intel VT-x Instruction Set Implementation

Mechanically in a Type 2 hypervisor scenario the instruction set works on a first-come first-served basis. Once VMM mode has been activated (using the `vmlaunch` instruction) the hypervisor has complete control over the system, even superseding the host operating system.

To initialize hypervisor code the following steps are required:

1. Check that a hypervisor isn't already running (optional)
2. Check that the Hardware and Firmware support VT-x
3. Allocate space for the `VMCS` (described later)
4. Capture register contexts (needed to support de-virtualizing safely)

5. Read and initialize multiple MSRs
6. Initialize and allocate EPT structures (if using)
7. Execute a `vmxon` instruction to enter VMX root mode
8. Set fields in the VMCS to dictate how the hypervisor should function and what conditions should cause VM exits
9. Execute a `vmlaunch` command to virtualize a processor

Note that the above steps need to be run for each processor on the system to prevent instability. Also note that the virtualization instructions are privileged and can only be executed in ring 0.

Recall that earlier we learned that a `vmcall` instruction causes a VM exit. A VM exit is an event that causes the CPU to transition from the Virtual Machine (VM) mode, where access to resources are controlled by the hypervisor, to Virtual Machine Monitor (VMM) mode, where the code executing has full control over the host operating system. The VMM is the heart of the hypervisor since it implements all the logic on how to manage resources on the system. The two terms will be used interchangeably for the remainder of this document.

The most crucial data structure when implementing a hypervisor is the Virtual Machine Control Structure (VMCS). This structure is treated as an opaque blob of memory accessed via the `vmwrite` and `vmread` instructions. Consulting the documentation[12] for one of these instructions shows that they take two operands: a field id and a value.

VMCS fields describe most aspects of the hypervisor including function pointers to call under different circumstances, conditions on when the CPU should exit to VMM mode, and metadata about the hypervisor. There are references online[13] for the full list of defined constants in the VMCS but the most important fields are:

1. `HOST_RIP` (0x6C16) – the address that should execute in VMM mode when a VM exit occurs
2. `HOST_RSP` (0x6C14) – the address of the VMM mode's RSP, set by the CPU when a VM exit occurs
3. `GUEST_RIP` (0x681E) – the address the guest (in VM mode) was executing when a VM exit occurred, and the address that should be executed when the CPU returns from VMM to VM mode
4. `GUEST_RSP` (0x681C) – the stack address of the guest (in VM mode) when a VM exit occurred, and the RSP address that should be set when the CPU returns from VMM to VM mode
5. `VM_EXIT_REASON` (0x4402) – allows the VMM to query why a VM exit occurred
6. `VM_EXIT_INSTRUCTION_LEN` (0x440C) – the length of the instruction that caused the VM exit

---

[12] https://www.felixcloutier.com/x86/VMWRITE.html

[13] https://developer.apple.com/documentation/hypervisor/1469436-virtual_machine_control_structur?language=objc

7. `EPT_POINTER` (0x201A) – a pointer that accepts a data structure describing the EPT structure for this hypervisor
    a. Fully documented in section 24.6.11 of Volume 3c of the Intel manual

This information gives us enough context to continue analyzing the challenge binary.

## Analysis of fhv.sys

`fhv.sys` is a driver loaded into ring 0, meaning we need to set up kernel debugging. Kernel debugging on windows is done using windbg. A full guide is not included with this walkthrough, but there are two main ways to connect a kernel debugger:

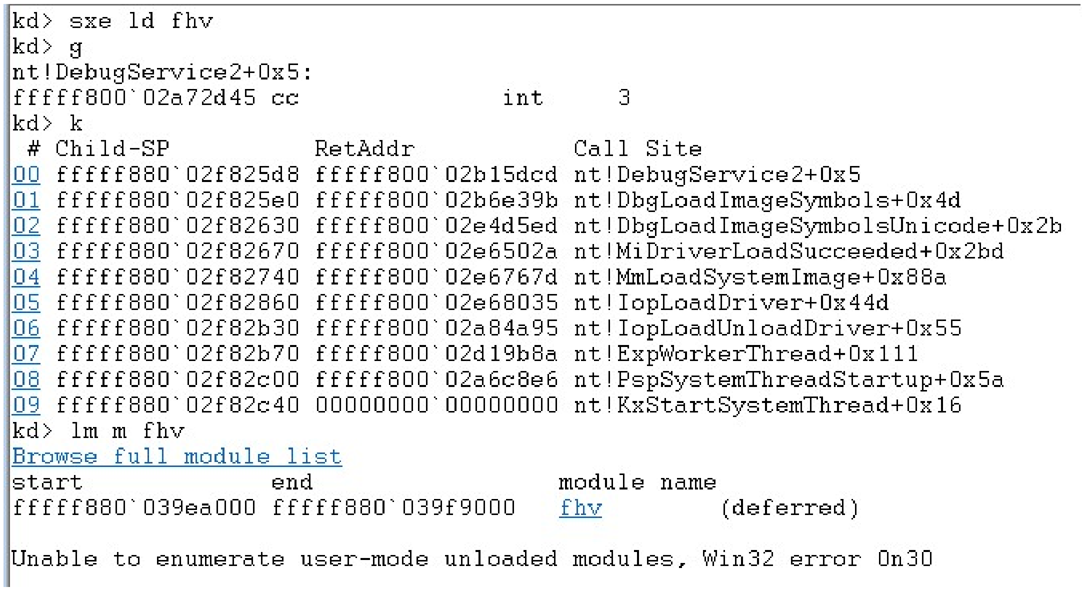
- Follow the steps outlined on MSDN[14] to debug over a (virtual) serial cable
- Use VirtualKD[15] which handles the debugger communication using custom binaries

Once our kernel debugger is connected we can use the `sxe` command to break when images (PE files) are loaded: the OS will break into the debugger when the image is loaded into memory. We can then rebase our IDA IDB and take a VM snapshot before continuing our analysis.

---

[14] https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/attaching-to-a-virtual-machine--kernel-mode-

[15] http://virtualkd.sysprogs.org/tutorials/install/

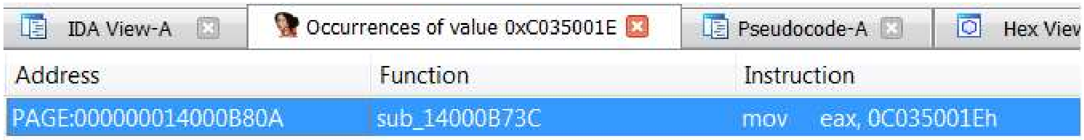


```
kd> sxe ld fhv
kd> g
nt!DebugService2+0x5:
fffff800`02a72d45 cc              int     3
kd> k
 # Child-SP          RetAddr           Call Site
00 fffff880`02f825d8 fffff800`02b15dcd nt!DebugService2+0x5
01 fffff880`02f825e0 fffff800`02b6e39b nt!DbgLoadImageSymbols+0x4d
02 fffff880`02f82630 fffff800`02e4d5ed nt!DbgLoadImageSymbolsUnicode+0x2b
03 fffff880`02f82670 fffff800`02e6502a nt!MiDriverLoadSucceeded+0x2bd
04 fffff880`02f82740 fffff800`02e6767d nt!MmLoadSystemImage+0x88a
05 fffff880`02f82860 fffff800`02e68035 nt!IopLoadDriver+0x44d
06 fffff880`02f82b30 fffff800`02a84a95 nt!IopLoadUnloadDriver+0x55
07 fffff880`02f82b70 fffff800`02d19b8a nt!ExpWorkerThread+0x111
08 fffff880`02f82c00 fffff800`02a6c8e6 nt!PspSystemThreadStartup+0x5a
09 fffff880`02f82c40 00000000`00000000 nt!KxStartSystemThread+0x16
kd> lm m fhv
Browse full module list
start             end                 module name
fffff880`039ea000 fffff880`039f9000   fhv        (deferred)

Unable to enumerate user-mode unloaded modules, Win32 error 0n30
```

**Figure 11 - Using the sxe command to start debugging a driver**

`DriverEntry` is the main entry point for .sys files. For this binary `DriverEntry` is at offset `0x1060`. This binary is large and based on the amount of logic that needs to run to virtualize a CPU, likely implements a lot of logic that is unlikely to be useful to analyze. Based on our earlier analysis we know that the entry point for the driver failed with status code `0xC035001E`. Searching for as an *immediate value* will show us where the driver sets this return code.



| Address | Function | Instruction |
|---|---|---|
| PAGE:000000014000B80A | sub_14000B73C | mov eax, 0C035001Eh |

**Figure 12 - Search results for our error code**

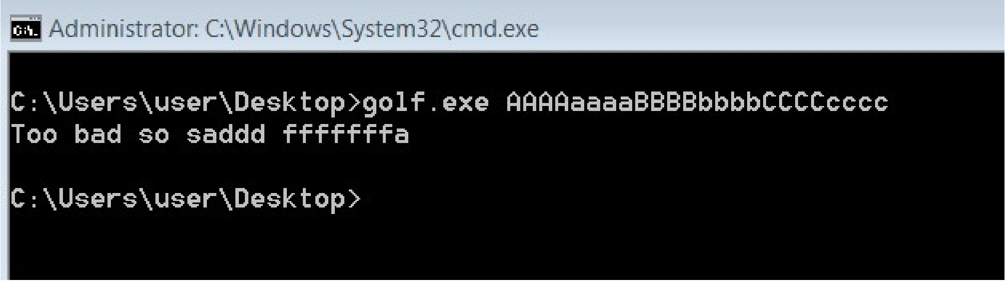
`sub_14000B73C` performs several checks can result in returning this error code. This function is performing the initialization checks detailed from steps 1 and 2 from the `Intel VT-x Instruction Set Implementation` section above. Debugging this function will reveal that the check that's failing due to the result of a cpuid instruction at offset `0xB777`. The driver tests if bit 5 in ECX is set, which consulting the documentation is checking if VMX is supported. In a default configuration of VMWare Workstation this bit is unset and the driver exits because VMWare is already virtualizing this system.

VMWare supports nested virtualization, which can be enabled by adding the following lines to your VMX file:

```
hypervisor.cpuid.v0 = "FALSE"
mce.enable = "TRUE"
vhv.enable = "TRUE"
```

**Figure 13 - Options to enable nested virtualization**

Restarting our VM after this allows us to load the driver successfully, which we can verify by rerunning `golf.exe` and checking the error code.

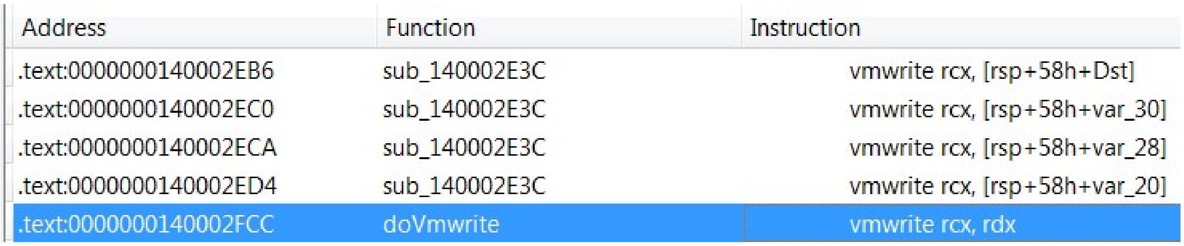


```
C:\Users\user\Desktop>golf.exe AAAAaaaaBBBBbbbbCCCCcccc
Too bad so saddd fffffffa

C:\Users\user\Desktop>
```

**Figure 14 - A new error code from running the executable**

Now that the challenge is running correctly we can revisit how the key is verified. Because most of the code is going to be standard hypervisor initialization code it is not beneficial for us to analyze it. However, since we know that the VMCS is crucial to VMM operation it may be useful to know what values are set so we know how the hypervisor is configured. Using the search function of IDA we can see that there are only two subroutines that use the `vmwrite` instruction, one of which looks like a utility function.



| Address | Function | Instruction |
|---|---|---|
| .text:0000000140002EB6 | sub_140002E3C | vmwrite rcx, [rsp+58h+Dst] |
| .text:0000000140002EC0 | sub_140002E3C | vmwrite rcx, [rsp+58h+var_30] |
| .text:0000000140002ECA | sub_140002E3C | vmwrite rcx, [rsp+58h+var_28] |
| .text:0000000140002ED4 | sub_140002E3C | vmwrite rcx, [rsp+58h+var_20] |
| .text:0000000140002FCC | doVmwrite | vmwrite rcx, rdx |

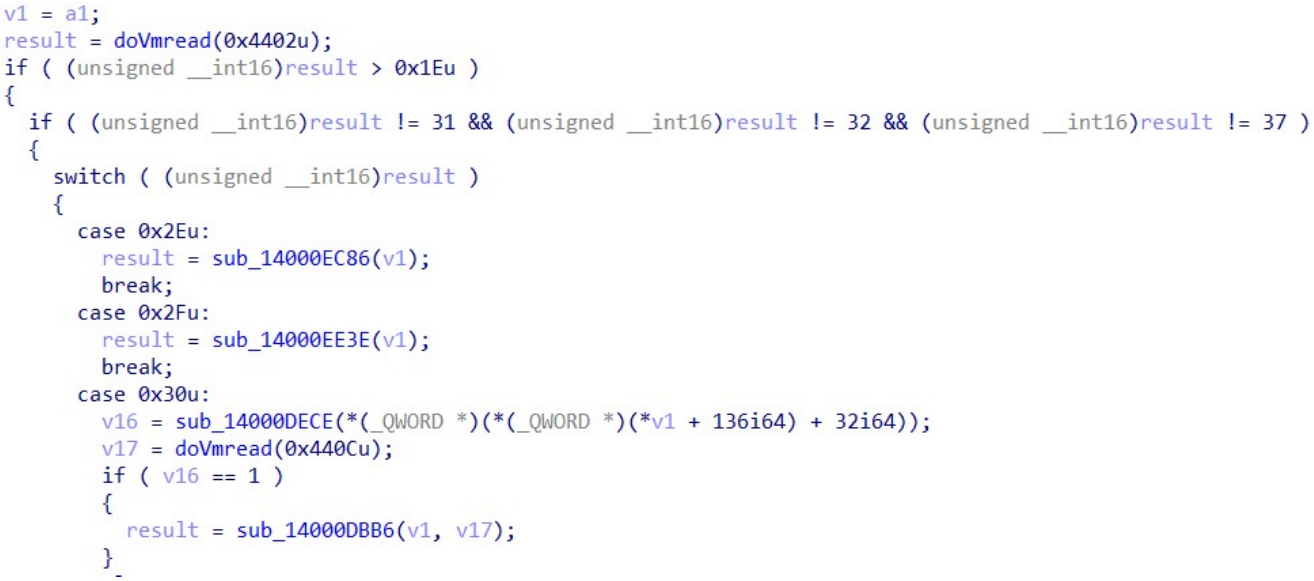
**Figure 15 - Text search for the vmwrite instruction**

The bp command in windbg can accept a series of commands to run when a breakpoint is hit. We can use this command to output all fields set in the VMCS. At a minimum, we want to record the value for `HOST_RIP` (0x6C16), which is the address where VMM execution begins when a VM exit happens. `The HOST_RIP` value for this binary is set to is `0x1400013E6.`

```
bp <vmwrite address> "r rcx; r rdx; g"
```

**Figure 16 - Breakpoint command to dump operands for vmwrite and continue**

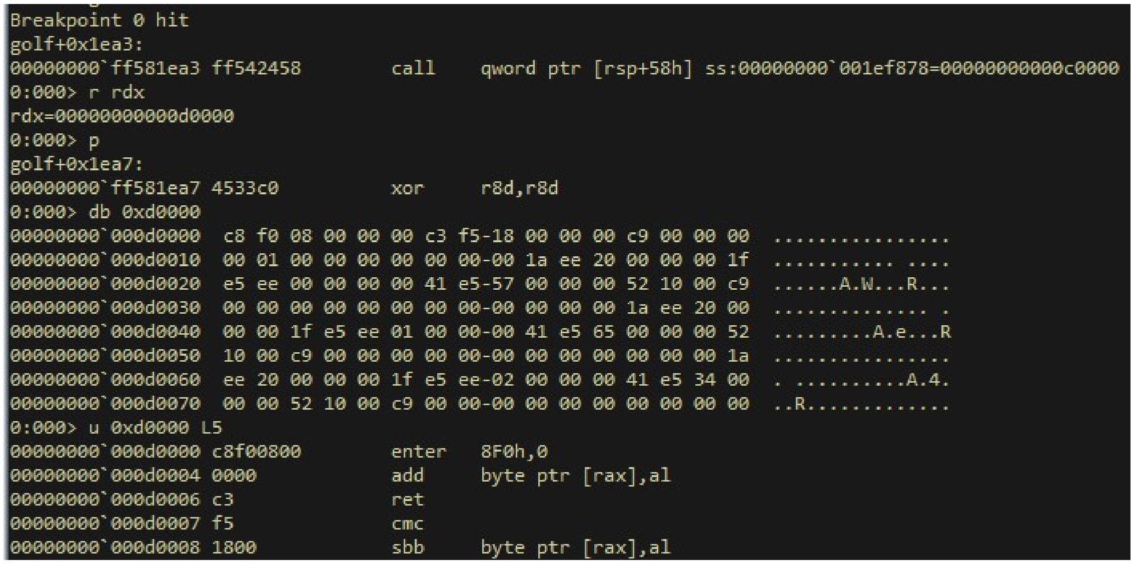
Since the only code we haven't fully analyzed in `golf.exe` is the functions performing `vmcall` instructions we should start with how the driver handles VM exits. Recall from the last section that the `VM_EXIT_REASON` VMCS field is used for the hypervisor to query why the CPU entered VMM mode. Repeating the immediate value search trick from Figure 15 shows us only one reference to that constant, which is passed into `sub_14000E7CA.` This function performs a `vmread` and returns the result.



```
v1 = a1;
result = doVmread(0x4402u);
if ( (unsigned __int16)result > 0x1Eu )
{
  if ( (unsigned __int16)result != 31 && (unsigned __int16)result != 32 && (unsigned __int16)result != 37 )
  {
    switch ( (unsigned __int16)result )
    {
      case 0x2Eu:
        result = sub_14000EC86(v1);
        break;
      case 0x2Fu:
        result = sub_14000EE3E(v1);
        break;
      case 0x30u:
        v16 = sub_14000DECE(*(_QWORD *)(*(_QWORD *)(*v1 + 136i64) + 32i64));
        v17 = doVmread(0x440Cu);
        if ( v16 == 1 )
        {
          result = sub_14000DBB6(v1, v17);
        }
```

Figure 17 - Decompilation of the VM_EXIT_REASON function

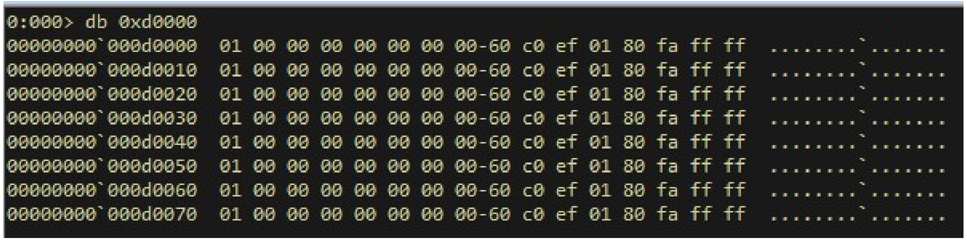
This function looks like the function that handles all supported VM exits. Since we've already crawled through enough documentation we can go back to golf.exe and set a breakpoint at the point that the `vmcall` instructions are executed. Since the usermode code looks like the `vmcall` instructions do something to populate the buffer that we end up executing, we can dump this data after the first `vmcall` (from the first of the four functions):



```
Breakpoint 0 hit
golf+0x1ea3:
00000000`ff581ea3 ff542458        call    qword ptr [rsp+58h] ss:00000000`001ef878=00000000000c0000
0:000> r rdx
rdx=00000000000d0000
0:000> p
golf+0x1ea7:
00000000`ff581ea7 4533c0          xor     r8d,r8d
0:000> db 0xd0000
00000000`000d0000  c8 f0 08 00 00 00 c3 f5-18 00 00 00 c9 00 00 00  ................
00000000`000d0010  00 01 00 00 00 00 00 00-00 1a ee 20 00 00 00 1f  ........... ....
00000000`000d0020  e5 ee 00 00 00 00 41 e5-57 00 00 00 52 10 00 c9  ......A.W...R...
00000000`000d0030  00 00 00 00 00 00 00 00-00 00 00 00 1a ee 20 00  .............. .
00000000`000d0040  00 00 1f e5 ee 01 00 00-00 41 e5 65 00 00 00 52  .........A.e...R
00000000`000d0050  10 00 c9 00 00 00 00 00-00 00 00 00 00 00 00 1a  ................
00000000`000d0060  ee 20 00 00 00 1f e5 ee-02 00 00 00 41 e5 34 00  . ..........A.4.
00000000`000d0070  00 00 52 10 00 c9 00 00-00 00 00 00 00 00 00 00  ..R.............
0:000> u 0xd0000 L5
00000000`000d0000 c8f00800        enter   8F0h,0
00000000`000d0004 0000            add     byte ptr [rax],al
00000000`000d0006 c3              ret
00000000`000d0007 f5              cmc
00000000`000d0008 1800            sbb     byte ptr [rax],al
```

Figure 18 - Dump of the data returned from the first vmcall

… which doesn't look like code, so we can dump the data after the second vmcall:



```
0:000> db 0xd0000
00000000`000d0000  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
00000000`000d0010  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
00000000`000d0020  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
00000000`000d0030  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
00000000`000d0040  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
00000000`000d0050  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
00000000`000d0060  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
00000000`000d0070  01 00 00 00 00 00 00 00-60 c0 ef 01 80 fa ff ff  ........`.......
```

**Figure 19 - Dump of the data returned from the second vmcall**

… which still doesn't look like code. Your analysis machine may perform slightly differently and show that the memory is inaccessible (all bytes show as `??`). Note that dumping the same address in the kernel debugger will show all zeroed data. Despite all of this, we can step over the call to execute this address without throwing any exceptions or crashing. While we're here, we can also verify that each of the four functions using the `vmcall` instruction accepts a position in the input string as a parameter:

- `sub_100001E40` accepts the command line string starting at position 0
- `sub_100001F20` accepts the command line string starting at position 5
- `sub_100002000` accepts the command line string starting at position 14
- `sub_1000020E0` accepts the command line string starting at position 19

It seems likely that these four subroutines run the algorithm to verify our key, but we need to understand the format of the data in this buffer to confirm.

The full list of VM exit reasons are listed in Appendix C of Volume 3D of the Intel Manual[16]. The `vmcall` instruction corresponds to exit reason 0x18 which in this driver calls `sub_140003810`. Consulting `golf.exe` we can see that:

- The second vmcall always uses the constant `0x13687451` as a parameter
- The third vmcall always uses the constant `0x13687453` as a parameter
- The first vmcall, which we saw returned unknown data to us, changes for each of the four functions

---

[16] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf

Each of our four `vmcall` functions uses a different constant as a parameter: `0x13687060`, `0x13687061`, `0x13687062`, `0x13687063`. Analyzing the `vmcall` handler shows a case statement switching on these constants, causing the hypervisor to copy one of four large buffers to a usermode address and XOR it with `0xE2`. The data post-XOR matches what was shown in the usermode debugger in Figure 18.

For vmcalls `0x13687451` and `0x13687453` the hypervisor resolves a physical address and calls the subroutine at address `0x1400026B0`. It is nonobvious what this routine does, performing some bit shifts and masks before returning a pointer to an unknown data structure.

We don't have any context for knowing what this function is doing. Based on other information we know, there are a few ways we can proceed with our analysis:

- We can trace all VM exits using the windbg command described in Figure 16 to see what happens during the rest of the execution
- We know that the buffer from `golf.exe` is executed somehow. There is a `GUEST_RIP` VMCS field that contains the instruction pointer when the VM exits or resumes execution. The hypervisor can update this field when handling and exit to skip over the instruction which caused the fault
    - We can use the VMCS field `GUEST_RIP` (0x681E) as a search term for a clue on where to look
- If we logged the `vmwrite` instructions earlier, we might notice that the parameter to `sub_1400042C8` matches what is written to the `EPT_POINTER` (0x201A) VMCS field
- Reverse where this data comes from. After tracing back to the HOST_RIP address (detailed below) we'll discover that this structure comes from the stack when a VM exit occurs.

These methods will eventually lead us to the subroutine at address `0x1400023AC`, which is a large switch statement. Each function in the switch statement sets data in an unknown data structure before adjusting the instruction pointer using a `vmwrite`. To understand what this function is doing we need to understand the structure being modified.

`sub_1400023AC` is eventually called from `sub_1400013E6.` There is another switch statement implemented in `sub_140004528` which switches on the result of a `vmread from VMCS` field `0x4402, VM_EXIT_REASON`. That means that `sub_140004528` contains all the logic for how this hypervisor should handle any VM exit.

The case statement that results in calling `sub_1400023AC` stems from a VM exit reason `0x30`, which is an EPT violation.

```
a1_2 = a1;
result = doVmread(0x4402u);                       // VM_EXIT_REASON
if ( (unsigned __int16)result > 0x1Eu )
{
  if ( (unsigned __int16)result != 31 && (unsigned __int16)result != 32 && (unsigned __int16)result != 37 )
  {
    switch ( (unsigned __int16)result )
    {
      case 0x2Eu:
        result = sub_14000347C(a1_2);
        break;
      case 0x2Fu:
        result = sub_140003634(a1_2);
        break;
      case 0x30u:
        v16 = sub_1400026C4(*(_QWORD *)(*(_QWORD *)(*a1_2 + 136i64) + 32i64));
        v17 = doVmread(0x440Cu);                 // VM_EXIT_INSTRUCTION_LEN
        if ( v16 == 1 )
        {
          result = sub_1400023AC(a1_2, v17);
        }
        else
        {
          result = (unsigned int)(v16 - 2);
          if ( (unsigned int)result <= 1 )
            result = sub_140002FCC(0x681Ei64, a1_2[2] + v17);// GUEST_RIP
        }
```

**Figure 20 - VM Exit handler function**

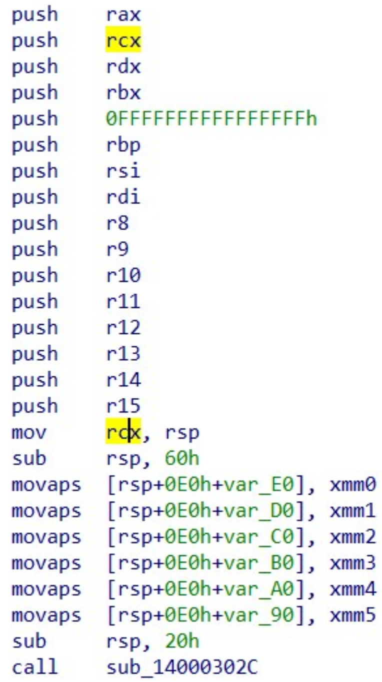
Extended Page Tables (EPT) is Intel's implementation of Second Level Address Translation[17] (SLAT). SLAT allows the hypervisor to allocate segments of virtual memory that pretend to be the physical addresses for a guest which is running in VM mode. EPT is supported in the hardware; when a guest accesses something it thinks is a physical address a VM exit automatically occurs and the hypervisor can work with the EPT structures to emulate the operation.

Because we know something is unusual with the memory in this buffer (not showing up as code after the first `vmcall`, not showing up as code or not at all after the second `vmcall`) and because we know the EPT_POINTER value is being used before the bitmask in `sub_0x1400026B0`, we can deduce that this routine is updating the page contents and permissions in the EPT structures. In the case of this binary this is modifying the permissions of this memory to be nonreadable, nonwritable, and nonexecutable. Because EPT, like the rest of a hypervisor's implementation, is meant to be invisible to the guest operating system, the guest can mark this page however it wants, but accessing the memory in a way that is noncompliant with the EPT permissions causes a VM exit with code `EPT_VIOLATION` (`0x30`). The hypervisor can then handle or modify the memory access according to its memory permissions.

The structure modified in `sub_1400023AC` is passed as a parameter starting with `sub_14000302C`. This subroutine is called by `sub_1400013E6` which was set as the `HOST_RIP VMCS` field. This function appears to be inline assembly which pushes all general-purpose registers and uses the stack location of these pushes as a parameter.

---

[17] https://en.wikipedia.org/wiki/Second_Level_Address_Translation



```
push    rax
push    rcx
push    rdx
push    rbx
push    0FFFFFFFFFFFFFFFFh
push    rbp
push    rsi
push    rdi
push    r8
push    r9
push    r10
push    r11
push    r12
push    r13
push    r14
push    r15
mov     rcx, rsp
sub     rsp, 60h
movaps  [rsp+0E0h+var_E0], xmm0
movaps  [rsp+0E0h+var_D0], xmm1
movaps  [rsp+0E0h+var_C0], xmm2
movaps  [rsp+0E0h+var_B0], xmm3
movaps  [rsp+0E0h+var_A0], xmm4
movaps  [rsp+0E0h+var_90], xmm5
sub     rsp, 20h
call    sub_14000302C
```

**Figure 21 - HOST_RIP entry point saving GP registers**

When a VM exit happens the only thing that the CPU does is redirect execution to the value of the HOST_RIP VMCS field and set the stack to the value of the HOST_RSP VMCS field. Everything else, including saving and restoring the guest context, is the hypervisor's responsibility. The parameter to `sub_14000302C` is a stack structure of the form:

```
typedef struct  _GUEST_REGISTERS
{
  ULONG_PTR r15;
  ULONG_PTR r14;
  ULONG_PTR r13;
  ULONG_PTR r12;
  ULONG_PTR r11;
  ULONG_PTR r10;
  ULONG_PTR r9;
  ULONG_PTR r8;
  ULONG_PTR rdi;
  ULONG_PTR rsi;
  ULONG_PTR rbp;
  ULONG_PTR rsp;
  ULONG_PTR rbx;
  ULONG_PTR rdx;
```

```
  ULONG_PTR rcx;
  ULONG_PTR rax;
} GUEST_REGISTERS, *PGUEST_REGISTERS;
```

**Figure 22 - Saved guest register state on vm exit**

After returning from `sub_14000302C` the registers are popped before a `vmresume` instruction is executed, returning to VM mode. The code in `sub_14000302C` uses a pointer to this structure and builds a new structure, which is passed down through the VM exit handling subroutines:

```
typedef struct _GUEST_CONTEXT
{
    GUEST_REGISTERS *gpRegs;
    PVOID guestFlags;                          // from vmread of GUEST_RFLAGS
(0x6820)
    PVOID guestRip;                               // from vmread of GUEST_RIP
(0x681E)
    PVOID currentIrql;
    PVOID currentIrql2;
    bool unknown;
} GUEST_CONTEXT, *PGUEST_CONTEXT;
```

**Figure 23 - Saved guest context on vm exit**

Returning to the `vmcall` handling function you might notice that the pointer being passed to the unknown parsing function is at offset `0x88` in the `GUEST_REGISTERS` structure, which is beyond the end of the data (and thus farther down the VMM stack) than we've reversed so far. To understand what's on the stack before saving the guest register state we would need to reverse how the `HOST_RSP` is set, which happens in the subroutine at address `0x14000BBCC`. Because we've already figured out that this structure is related to EPT data this is left as an exercise to the reader.

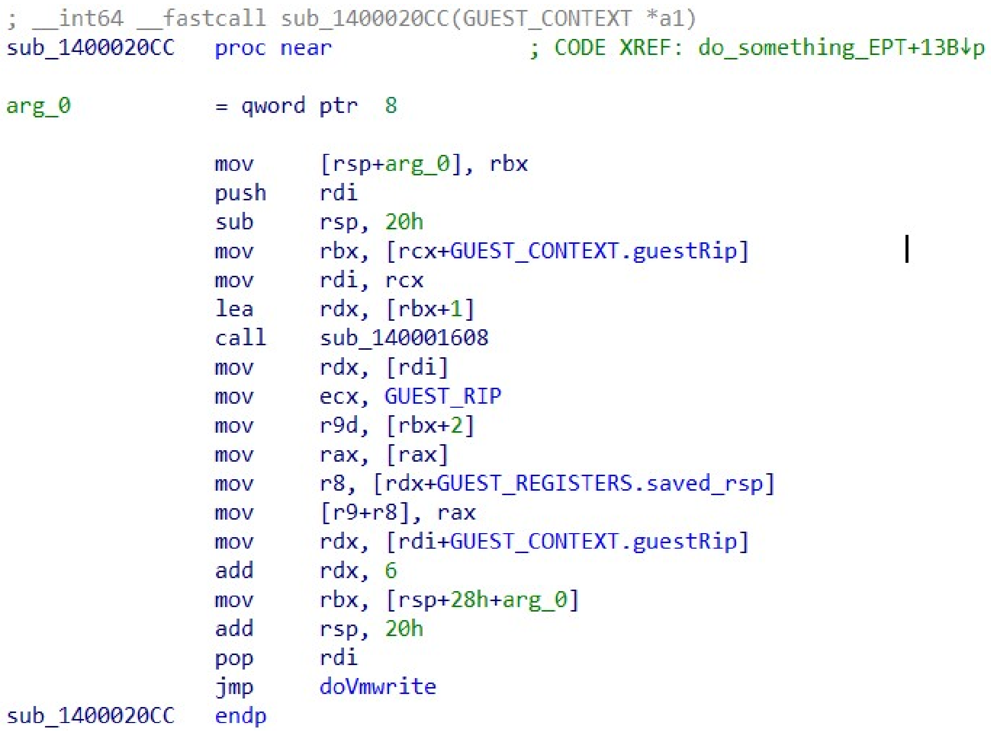
Now that we know the data being passed through the VM exit handling code we can return to the large switch statement from before. Note that if you found this function earlier in your analysis you could skip/ignore most of the machinations of how the instruction set works and only focus on the code path leading here.

```
__int64 __fastcall do_something_EPT(GUEST_CONTEXT *a1)
{
  GUEST_CONTEXT *v1; // rbx
  __int64 v2; // rax
  __int64 v3; // rdx
  unsigned __int64 v4; // rdi
  char *v5; // rcx
  char v6; // al
  char v7; // cl

  v1 = a1;
  v2 = doVmread(0x6802u);                      // GUEST_CR3
  v4 = __readcr3();
  __writecr3(v2);
  v5 = (char *)v1->guestRip;
  v6 = *v5;
  if ( *v5 == 1 )
  {
    sub_140001FAC((__int64)v1);
    goto LABEL_91;
  }
  switch ( v6 )
  {
    case (char)0xBB:
      sub_140001EBC(v1);
      break;
    case (char)0xAA:
      sub_140001F18(v1);
      break;
    case (char)0xC2:
      sub_1400016EC(v1);
      break;
```

Figure 24 - Decompilation of sub_1400023AC

This subroutine reads one byte from the address causing the EPT violation and performs different functionality depending on its values. Recall from Figure 18 the first byte we know was returned to `golf.exe` and executed is `0xC8`, which results in calling `sub_1400020CC`.



```
; __int64 __fastcall sub_1400020CC(GUEST_CONTEXT *a1)
sub_1400020CC   proc near               ; CODE XREF: do_something_EPT+13B↓p

arg_0           = qword ptr  8

                mov     [rsp+arg_0], rbx
                push    rdi
                sub     rsp, 20h
                mov     rbx, [rcx+GUEST_CONTEXT.guestRip]
                mov     rdi, rcx
                lea     rdx, [rbx+1]
                call    sub_140001608
                mov     rdx, [rdi]
                mov     ecx, GUEST_RIP
                mov     r9d, [rbx+2]
                mov     rax, [rax]
                mov     r8, [rdx+GUEST_REGISTERS.saved_rsp]
                mov     [r9+r8], rax
                mov     rdx, [rdi+GUEST_CONTEXT.guestRip]
                add     rdx, 6
                mov     rbx, [rsp+28h+arg_0]
                add     rsp, 20h
                pop     rdi
                jmp     doVmwrite
sub_1400020CC   endp
```

**Figure 25 - Function handling byte 0xC8**

This subroutine:

1. Reads the instruction pointer which caused the EPT violation
2. Takes the second byte at this address and passes it as a parameter to `sub_140001608`
3. Dereferences the return from this function and stores the value at `GUEST_RSP` plus the DWORD value starting at byte 3 from the violation address
4. Increments the `GUEST_RIP` VMCS field by 6
    - This means when we re-enter VM mode RIP is 6 bytes forward

`sub_140001608` is called by most of the functions in the EPT switch routine; this implementation is the last piece of information we need to know to solve the challenge. This function takes the `GUEST_CONTEXT` structure and the second byte from the violation address as parameters. Based on the value of this byte, the subroutine returns the address of a field in the `GUEST_REGISTERS` structure.

The first 6 bytes from Figure 6 are: `0xC8 0xF0 0x08 0x00 0x00 0x00`. Based on the previous two subroutines these bytes are interpreted by the hypervisor as:

1. `0xC8` – call `sub_1400020CC` which stores some value at some offset from `GUEST_REGISTERS->rsp`
2. `0xF0` – used as an index by `sub_140001608` to return the address of `GUEST_REGISTERS->rcx`
3. `0x08 0x00 0x00 0x00` – treated as a DWORD (8) and added to `GUEST_RSP->rsp` to determine the address to store the data

This means that to the hypervisor the byte sequence `0xC8 0xF0 0x08 0x00 0x00 0x00` is equivalent in the guest VM to `mov [rsp+8], rcx` !

The subroutine at address `0x1400023AC` implements a virtual instruction set. The first byte at the guest's instruction pointer designates a different translation of an x86 opcode. The data buffers returned from vmcalls `0x13687060`, `0x13687061`, `0x13687062`, and `0x13687063` return sequences of these virtual opcodes and these opcodes implement four different algorithms which verify the key passed on the command line. In order to solve the challenge we need to reverse the virtual opcodes, convert them to x86, and then solve each of the four subroutines validating the key.

We now understand the pieces necessary to solve this challenge. To recap:

- `golf.exe` accepts a 24-character key on the command line and drops and loads fhv.sys
- `fhv.sys` (if the system supports it) virtualizes the operating system
- `golf.exe` runs four subroutines to verify different parts of the key. Each subroutine
  - Calls `vmcall` with a different constant to get a buffer of virtualized opcodes
  - Uses vmcall with constant `0x13687451` to tell the hypervisor to mark the buffer from the previous `vmcall` as EPT non-executable
  - Executes the virtual opcodes
- When `golf.exe` attempts to execute an EPT-protected page the hypervisor takes over
  - A VM exit is cause with VM_EXIT_REASON EPT_VIOLATION
  - This results in calling the subroutine at address `0x1400023AC` in `fhv.sys`
  - This subroutine reads the violation address, treats the first byte as a virtual opcode, and processes the virtual instruction by modifying fields in the `GUEST_REGISTERS` structure and updating the `GUEST_RIP VMCS` field
  - After handling the exit, the modified registers are restored and `vmresume` is called to return to the next virtual instruction
    - Note that some opcodes that modify the stack or instruction pointer use `vmwrites` to change the guest state

The Appendices give the full list of virtual opcodes supported by `fhv.sys`, along with the original source code for the four algorithms validating the key. Once the opcodes are known the algorithms are simple and reversing them will yield the key: `We4r_ur_v1s0r_w1th_Fl4R3@flare-on.com`

# Appendix 1: Full Virtual Opcode List

| Virtual Opcode | x86 Equivalent | Virtual Opcode Size |
|---|---|---|
| 0 | mov <reg64>, <reg64> | 3 |
| 1 | ret | 1 |
| 2 | mov <reg32>, <reg32> | 3 |
| 0x17 | mov [rsp+<offset>], al | 5 |
| 0x19 | mov <reg32>, [rsp+<offset>] | 6 |
| 0x1A | mov <reg64>, [rsp+<offset>] | 6 |
| 0x1B | mov al, [rsp+<offset>] | 5 |
| 0x1C | movzx <reg32>, [rsp+<reg>…+<offset>] | varies |
| 0x1D | movsxd rax, [rsp+<reg64>…+<offset>] | varies |
| 0x1E | movsx <reg32>, [rsp+<reg>…+<offset>] | varies |
| 0x1F | movsx <reg32>, [<reg64>+<reg64>] | varies |
| 0x20 | movzxd <reg64>, [<reg64>+<reg64>...] | varies |
| 0x30 | rep stosb | 1 |
| 0x40 | cmp <reg32>, <reg32> | 3 |
| 0x41 | cmp <reg32>, <val> | 6 |
| 0x42 | cmp [<reg32>+<offset>], <val> | 9 |
| 0x43 | test <reg64>, <reg64> | 3 |
| 0x44 | test <reg32>, <reg32> | 3 |
| 0x4A | lea <reg64>, [rsp+<offset>] | 6 |
| 0x4B | lea <reg64>, [rsp+<offset>] | 6 |

| | | |
|---|---|---|
| 0x50 | jmp | 3 |
| 0x51 | jnz | 3 |
| 0x52 | jz | 3 |
| 0x53 | jbe | 3 |
| 0x54 | jge | 3 |
| 0xAA | push <reg> | 2 |
| 0xB9 | shl <reg64>, <val> | 3 |
| 0xBA | shl <reg32>, <val> | 3 |
| 0xBB | pop <reg> | 2 |
| 0xBC | shr <reg32>, <val> | 3 |
| 0xBD | and <reg64>, <val> | 10 |
| 0xBE | and <reg32>, <val> | 6 |
| 0xBF | xor <reg64>, <val> | 10 |
| 0xC0 | xor <reg32>, <val> | 6 |
| 0xC1 | add <reg64>, <val> | 6 |
| 0xC2 | add <reg64>, reg64> | 6 |
| 0xC3 | sub <reg64>, <val> | 6 |
| 0xC4 | sub <reg64>, reg64> | 6 |
| 0xC5 | xor <reg64>, <reg64> | 6 |
| 0xC6 | mov <reg64>, <val> | 10 |
| 0xC7 | mov [<reg64>], <reg64> | 3 |
| 0xC8 | mov [rsp+<offset>], <reg> | 6 |

| | | |
|---|---|---|
| 0xC9 | mov [rsp+<offset>, <val> | 13 |
| 0x1D | movsxd <reg64>, [rsp+<offset>] | Varies |
| 0xD1 | add <reg32>, <val> | 6 |
| 0xD2 | add <reg32>, <reg32> | 6 |
| 0xD3 | sub <reg32>, <val> | 6 |
| 0xD4 | sub <reg32>, <reg32> | 6 |
| 0xD5 | xor <reg32>, <reg32> | 3 |
| 0xD6 | mov <reg32>, <val> | 6 |
| 0xD7 | xor <reg32>, [rsp+<offset>] | 6 |
| 0xD8 | mov [rsp+<offset>], <reg32> | 6 |

# Appendix 2: Source code for key algorithm 1

```
//part 1 - static string cmp "We4r_"

BOOLEAN crackmePart0(char *password)
{
    BOOLEAN retVal = TRUE;
    if(password[0] != 0x57)
    {
        retVal = FALSE;
    }
    if(password[1] != 0x65)
    {
        retVal = FALSE;
    }
    if(password[2] != 0x34)
    {
        retVal = FALSE;
    }
    if(password[3] != 0x72)
    {
        retVal = FALSE;
    }
    if(password[4] != 0x5f)
    {
        retVal = FALSE;
    }
    return retVal;
}
```

## Appendix 3: Source code for key algorithm 2

```
//part 2 - XOR match "ur_v1s0r_"
BOOLEAN crackmePart1(char *password)
{
    BOOLEAN retVal = TRUE;
    char keyBuf[9];
    keyBuf[0] = '\x0';
    keyBuf[1] = '\x7';
    keyBuf[2] = '\x2a';
    keyBuf[3] = '\x3';
    keyBuf[4] = '\x44';
    keyBuf[5] = '\x6';
    keyBuf[6] = '\x45';
    keyBuf[7] = '\x7';
    keyBuf[8] = '\x2a';
    unsigned char xorKey = '\x75';

    for(int i = 0; i < 9; i++)
    {
        if((password[i] ^ xorKey) != keyBuf[i])
        {
            retVal = FALSE;
        }
    }

    return retVal;
}
```

# Appendix 4: Source code for key algorithm 3

```
// part 2 – “w1th_”  rolling XOR
BOOLEAN crackmePart2(char * password)
{
    BOOLEAN retVal = TRUE;
    char key = '\x80';
    //char modifier = '\x52';
    char keyBuf[5];
    keyBuf[0] = '\xa5';
    keyBuf[1] = '\xb1';
    keyBuf[2] = '\x02';
    keyBuf[3] = '\x4c';
    keyBuf[4] = '\xc5';


    for(int i = 0; i < 5; i++)
    {
        char tmpVal = (password[i] ^ key) ^ 0x52;
        if(tmpVal != keyBuf[i])
        {
            retVal = FALSE;
        }
        key = key + 0x52;
    }

    return retVal;
}
```

## Appendix 5: Source code for key algorithm 4

```
// part 4 - "CRC" brute force Fl4R3
BOOLEAN crackmePart3(char * password)
{
    BOOLEAN retVal = TRUE;
    ULONG sum = 0;
    int i = 0;

    sum = sum + password[0] + password[1] + password[2] + password[3] + password[4];

    retVal = TRUE;

    if(password[0] != 0x46)
    {
        retVal = FALSE;
    }

    if(password[4] != 0x33)
    {
        retVal = FALSE;
    }

    if(0x16b != sum)
    {
        retVal = FALSE;
    }

    if((password[2] + password[3]) != 0x86)
    {
        retVal = FALSE;
    }

    if((password[1] + password[2]) != 0xa0)
    {
        retVal = FALSE;
    }
    return retVal;
}
```