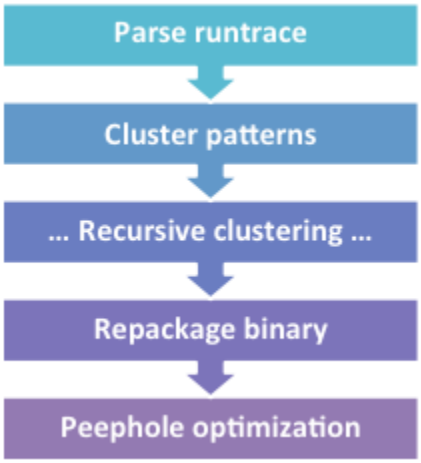
Background

The **Virtual Deobfuscator** was developed as part of the DARPA Cyber Fast Track program. The goal was to create a tool that could remove virtual machine (VM) based protections from malware. We developed a prototype version that looks very promising.

Virtual machine protections are a relatively new form of obfuscation. They work by translating sections of a binary's original machine code into bytecode for a custom VM. This transformation is destructive — the original binary is lost. The VM itself is embedded in the protected binary. It is used at runtime to interpret the instructions that were converted to bytecode.

The goal of the Virtual Deobfuscator is to analyze a runtrace and filter out the VM processing instructions, leaving a reverse engineer with a bytecode version of the original binary. It doesn't need to be tailored to the particular VM being analyzed, and so far it's worked on all the VM interpreters we've tested it on.

How it works



The Virtual Deobfuscator is based on pattern matching. It will analyze a runtrace and match patterns of instructions called *clusters*. This process continues recursively until no more instructions or clusters can be grouped into larger clusters. The remaining unclustered instructions contain the interpreted bytecodes; they are the instructions actually executed by the VM as it processed bytecodes.

Since protection VMs generally use RISC-based architectures, their instruction sets are simpler. This means that most instructions from the original program are represented by multiple bytecodes. The post-clustering instruction trace, then, contains a lot more instructions than the original binary did. To clean it up, we run the instructions through a peephole optimizer to remove redundant instructions and get something closer to the original.

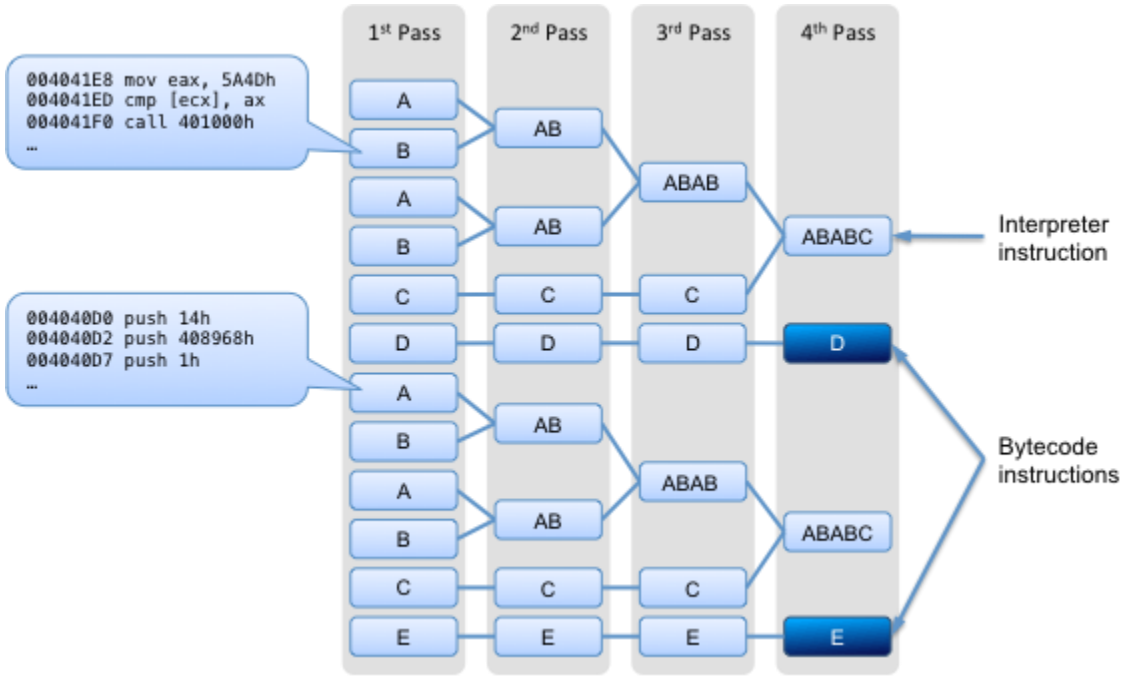
Parsing

The Virtual Deobfuscator's parser can handle traces from three popular debugging tools: WinDbg, OllyDbg, and Immunity Debugger. It can easily be extended to work with traces generated by other tools.

The parser converts traces to a normalized XML format for later processing, so tool developers can also modify their tools output directly to that format. There's no DTD for our XML format, but it's extremely simple. The short document below describes the format.

```
<root>
  <ins>0
    <thread>main</thread>
    <va>00432213</va>
    <mnem>JMP 0043A73A</mnem>
    <regs>None</regs>
  </ins>
  <ins>1
    <thread>main</thread>
    <va>0043A73A</va>
    <mnem>PUSH OFFSET 0043A6A3</mnem>
    <regs>[012FF84]=0 ESP=0012FF84</regs>
  </ins>
<root>
```

Clustering



Our theory was that since VMs work by interpreting bytecodes, there will be a lot of repeated groups of instructions that can be filtered out of a runtrace. The Virtual Deobufscator's clustering algorithm works by matching patterns of instruction sequences from a runtrace and grouping them into clusters. This process is repeated recursively, with new clusters being formed from free instructions and/or existing clusters, until no new clusters can be formed. At that point we're left with the interpreted bytecodes of the original program. In other words, the non-clustered instructions that are left are those that were executed when the VM was interpreting bytecodes.

Consider the figure above. Each single-letter block denotes one or more instructions that form a cluster. Multi-letter blocks represent clusters made up of other clusters. In the "1st Pass" column there are four "A" clusters, each having the same set of instructions executing at the same virtual addresses. Register differences are ignored when forming clusters. During each subsequent pass, the current set of instructions and/or clusters is examined, and new clusters are formed where possible.

Clusters are named according to a standard convention. For example, `c2_____#8`:

- **c** — the processing round ("a", "b", "c", etc.) *[c = round 3]*
- **2** — ascending integer, unique per round *[ID = 2]*
- _____ — an indicator of how many instructions are in the cluster; the more '_' (and the longer the cluster name), the more instructions are in the cluster
- **#8** — number of instructions in the cluster *[size = 8]*

Example

Below is pseudocode for a simple program we'll use to show how this works:

```
main()

  Loop 1

    Loop 2

      if (only)

        _asm { mov eax, 0xDEADBEEF }

        only = false
```

This example illustrates the principle of VM operation we're interested in — namely, that there are sets of repeating instructions (Loop 1 and Loop 2) with individual instructions popping up every once in a while (when the condition is true, in this case).

Assuming we had a runtrace of this code in file `runtrace.txt`, we would import it with the command:

```
python VirtualDeobfuscator.py -i runtrace.txt -d 1
```

Once the file was imported, we create clusters with the command:

```
python VirtualDeobfuscator.py -c -d 1
```

This command will generate a number of files, but the only two that are really of interest are:

- `[last round]_cluster.txt` — the cluster file generated during the final round ("a", "b", etc.) of clustering; it includes clusters and the virtual addresses of non-clustered instructions
- `final_assembly.txt` — similar to the final round cluster file, but it includes full assembly instructions as well as virtual addresses for non-clustered instructions

Below is a fragment of the `final_assembly.txt` file:

```
... (start up code)

004113D3   jmp short 004113DE

c1_____#11

c2_____#8

f1_____#47

c1_____#11

a21_#2

c2_____#8
```

```
a21_#2

00411411   mov eax,DEADBEEF   ;eax=DEADBEEF

f1____________#47

a16_#2

00411427   mov esi,esp   ;esi=0018FE34

... (wrap up code)
```

The fragment above starts with the jump to the program's `main` function. Our clustering algorithm has grouped all of the loop management code (conceptually similar to a VMs interpreter code) into just a few clusters. This makes the instruction that is not part of the loop management code (in blue) really stand out.

Binary repackaging

The repackaging step uses the output of the clustering process to create a binary fragment containing the original x86 program code without the VM. This allows for further analysis in disassemblers such as IDA Pro. We generate the binary by assembling the “sections” of assembly code created by the Virtual Deobfuscator. This code is stored in the file `final_assembly_nasm.asm`. We assemble it using the Netwide Assembler (NASM).

Peephole optimization

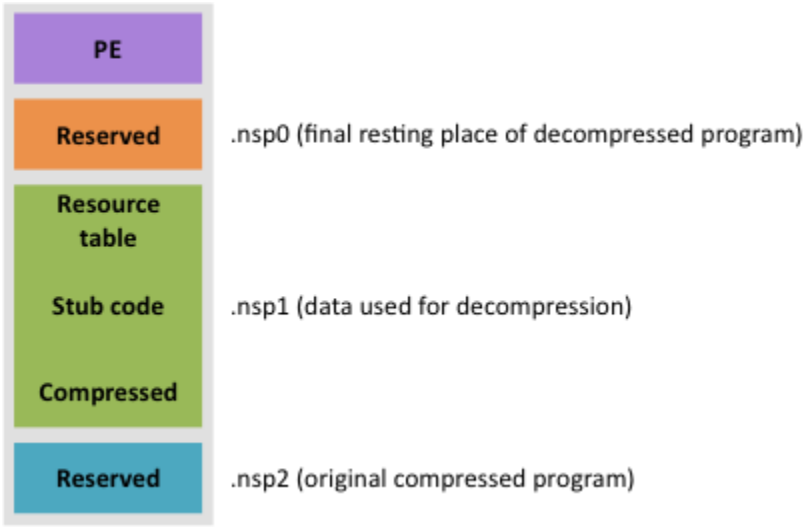
Once the runtrace has been reduced to just the bytecode instructions and packaged as a binary, we run the code through a peephole optimizer, implemented as an IDA Pro Python script. This will take care of any remaining redundancy in the code (remember, the bytecodes are for a RISC machine, so there will be redundancy compared to the original CISC instructions). This step also help remove simple obfuscations.

In the real world

We tested the Virtual Deobfuscator on the malware **Win32.Klone.af**, available from the Open Malware library. Klone uses both VM obfuscation, in the form of VMProtect, and a general protection packer, NSPack. Both of these protections are commercially available tools.

FreeSpyCheck.com provides the following description of Win32.Klone.af:

Packed.Win32.Klone.af is a memory resident Trojan that can attach itself to explorer.exe, track your keystrokes and communicate with precarious outlying servers. The trojan may randomly displays ads and false warnings on your computer. Upon installation, Packed.Win32.Klone.af may generate corrupt files, disable security programs and produce irritating popups. Packed.Win32.Klone.af may redirect your personal information and harm or alter important system files.



Klone was initially wrapped with NSPack. This creates 3 sections in a PE file: one containing a compressed and encrypted version of the original program (.nsp2), one containing resources used to decrypt the program (.nsp1), and one to hold the program once it's been unpacked (.nsp0).

Some of the information required to unpack the program is added to .nsp1 at runtime. VMProtect was applied on top of NSPack to hide this process, breaking existing NSPack unpacking tools. In particular, VMProtect was used to obfuscate the following steps in the unpacking process:

- decryption of the compressed section of .nsp1
- initialization of local variables for `VirtualAlloc`
- dynamic memory allocation (`VirtualAlloc`)
- finalization of the resource table in .nsp1

Results

We didn't spend much time reversing NSPack or the malware itself in this case — we were only really interested in seeing how the Virtual Deobfuscator could do with the VMProtected parts. Without the source code for the malware we can't be sure we fully extracted all the original program instructions. However, Virtual Deobfuscator was able to strip a sizeable amount of code away in a relatively short time, which made our analysis much faster.

Contact

Jason Raber
jason.raber@hexeffect.com
HexEffect, LLC
www.hexeffect.com