27- More AD and Anti-Debugging

Misusing Structured Exception Handling
SEH Chain
Thwarting Stack-Frame Analysis
Anti-Disassembly Recap
Anti-Debugging
Windows Debugger Detection
Identifying Debugger Behavior
Interfering with Debugger Functionality
Debugger Vulnerabilities
Debugging Demo
Misusing Structured Exception Handling

- Recall that SEH is used for error handling.
- Exceptions may occur because of issues like
  - Invalid memory region accesses
  - Divide by zero
  - Calls to RaiseException
- Head of the SEH Chain is referenced in the FS register, which points at the Thread Environment Block (TEB). First element of TEB is Thread Information Block (TIB). First element of TIB is SEH chain (linked list of 8-byte structure).
SEH Chain

- Struct contains two DWORD pointer elements
  
  ```
  DWORD prev;
  DWORD handler;
  ```

- Add a handler MyExceptionHandler to the chain as follows
  
  ```
  push MyExceptionHandler
  push fs:[0]
  mov fs:[0], esp
  ```

- Now MyExceptionHandler is at the head of the SEH chain and will be called first if an exception is raised. Can arrange this easily by generating divide by 0.

- This only works if program satisfies Software Data Execution Prevention (DEP). Methods to override this exist.

- Can easily restore SEH to original state.
Thwarting Stack-Frame Analysis

- Correct interpretation of the stack frame makes program analysis easier.
- Defeating Hex-Rays decompiler IDA plug-in, especially hurts analysis. (Ask Mr. Palecek.)
- Code shown on page 347 adjusts the stack pointer based on whether or not $\text{esp} < 1000h$ (which is almost always false). On one branch (the $\geq$ branch) $\text{esp}$ is incremented by 4. On the other branch (which IDA apparently trusts) $\text{esp}$ is incremented by 104h.
- This makes IDA think the program has 62 arguments.
- This esp check is almost always going to be false, but IDA can't check it like it might with a comparison to a register that was just zeroed. It takes more knowledge to realize this is really constant.
Anti-Disassembly Recap

- Disassemblers must make assumptions. Any assumption a disassembler makes may be used against it.
Anti-Debugging

• If malware can detect the presence of a debugger, it can modify its control flow to make analysis more complicated.
Windows Debugger Detection - 1

• Windows API Methods
  - Function `CheckRemoteDebuggerPresent` can check the `IsDebugged` field in current process or another process (by handle).
  - Function `NtQueryInformationProcess` in Ntdll.dll can check ProcessDebugPort.
  - Function `OutputDebugString` will succeed if a debugger is attached and return an error code otherwise.
Windows Debugger Detection - 2

- **Manual Structure Checks**
  - BeingDebugged flag (offset 2) in Windows PEB structure (at fs: [30h]). Can be manually reset.
  - ProcessHeap (offset 0x18) in Windows PEB contains ForceFlags and Flag fields (offset depends on OS). Can modify manually or set windbg -hd flag.
  - NTGlobalFlag PEB offset 0x68 has value 0x70 is a debugger is running. Same countermeasure as ProcessHeap.

- **System Residue**
  - Debugger may set resource HG\CurrentVersion\AeDebug
  - Debugger process may have an identifiable name (OllyDbg)
Identifying Debugger Behavior - 1

- **INT Scanning**
  Malware can look for 0xCC \((\text{int } 3)\) or 0xCD \((\text{int immediate})\) within the process's code to find breakpoints. This can be overcome by using hardware breakpoints.

- **Code checksums**
  Malware can perform a checksum (CRC, MD5) of its code to detect modifications. This can be overcome by using hardware breakpoints or changing the control flow of the inspecting code.

- **Timing checks (popular)**

- **Record a timestamp of the program and then record another. If the lag is too great debugging may be taking place. Or record timestamp before an exception then see how long handling takes. Debugger will generally require human intervention (lots of time).**
Identifying Debugger Behavior - 2

• Timing inquiry methods:
  - Use `rdtsc` instruction (0x0F31). Stores ticks since last reboot in `edx:eax`.
  - Function `QueryPerformanceCounter` yields a 1μsec clock value.
  - Function `GetTickCount` returns number of msecs since the system was rebooted.

• Overcome by identifying the successive calls to the timing check then modifying the result.
Interfering with Debugger Functionality - 1

• Thread Local Storage (TLS) callbacks execute *before* the entry point! Thus that code is executed before the debugger gains control. To use such callbacks, code usually has a .tls section. If it exists, suspect anti-debugging.
  
  – In IDA ctrl-E displays all entry points including TLS callbacks.
  – In OllyDbg, use *Options->DebuggingOptions->Events* to make first pause at *System Breakpoint*. 
Interfering with Debugger Functionality - 2

• Exceptions
  If trapped in debugger, malware may detect that normal handling doesn't occur.

• Inserting Interrupts
  – Can thwart analysis by causing debugger to pause unexpectedly.
  – int 3 insertion can cause WinDbg to repair code incorrectly.
  – int 2dh is method for kernel debugger to set breakpoint. (Missing Listing 16-10.)
  – icebp, In-Circuit Emulator breakpoint instruction. If step over this, debugger will think this is a normal single-step and will not execute the appropriate exception handler. Don't step over icebp!
Debugger Vulnerabilities

- OllyDbg will crash if PE Header is appropriately altered.
  - `NumberOfRvaAndSizes` (fixed at 0x10) entry is ignored by Windows loader, but extremely large (incorrect) values will crash Olly.
  - If a section's `SizeofRawData` is huge and `VirtualSize` is small Olly will crash because it allocates space based on `SizeofRawData`, but Windows loader uses `VirtualSize`. 
Debugging Demo

• Debug brbbot
• Open Names Window
• Look for CryptDecrypt
• Set hardware breakpoint on access
• Run program
  – Set software breakpoint on return address/continue
  – Set hardware breakpoint on return address/continue
• Rollback to snapshot, debug brbbbot
• Open Names Window
• Look for ReadFile
• Set breakpoint on all references
• Identify handle in stack
  – Set software breakpoint to reference in stack/continue
  – Set hardware breakpoint to reference in stack/continue
Next Time

• Read PMA Chapter 17