Improving Software Security with Dynamic Binary Instrumentation

Richard Johnson (rjohnson@sourcefire.com)
Principal Research Engineer Sourcefire VRT
The Good

- Software vulnerability mitigations are an effective approach at making exploitation more { difficult | expensive | ineffective }

- Mitigations have been developed for most major memory-related vulnerability classes
The Bad

- Due to the difficulty of development, mitigations are almost exclusively developed by vendors (with a few short-lived exceptions).

- Vendors supply mitigation technologies but do not enforce their use by 3rd party developers.
The Ugly

- Understanding and defeating mitigations are a top priority for vulnerability researchers regardless of domain

- Current vendor mitigations are defeated by modern exploitation techniques
The Challenge

- Determine if current binary instrumentation frameworks provide the required technology to develop one-off custom mitigations

Criteria
- Stability
- Speed
- Ease of implementation
DYNAMIC BINARY INSTRUMENTATION
Dynamic Binary Instrumentation (DBI) is a process control and analysis technique that involves injecting instrumentation code into a running process.

- DBI can be achieved through various means:
  - System debugging APIs
  - Binary code caching
  - Virtualization / Emulation
DBI Frameworks

- A DBI Framework facilitates the development of Dynamic Binary Analysis (DBA) tools

- DBI Frameworks provide an API for binary loading, process control, and instrumentation
  - DynamoRIO
  - PIN
  - Valgrind
DBI Architecture

Executing Process

DBI Framework

Transform
Profile
Cache
Execute

Operating System / Hardware

Plugins

Analysis & Mitigations

Instrumentation APIs
PIN Architecture

DIAGRAM CONTENT:
- Launcher Process
  - PIN.EXE
  - Launcher
- Server Process
  - PIN.EXE
  - Injection helper
  - Symbol server
  - DBGHELP.DLL
- Application Process
  - Application Code and Data
  - PINTOOL.DLL
  - PIN.LIB
  - Shared memory
- Virtual Machine Monitor
  - PINVM.DLL
  - System Call Emulator
  - Event Dispatcher
  - Thread Dispatcher
  - System Gate
- Code Cache
- Kernel Libraries
  - KERNEL32.DLL
  - NTDLL.DLL
- Windows Kernel

NOTES:
- The diagram illustrates the architecture and interaction between different components of PIN.
- Key interactions include CreateProcess, startup time, and shared memory.
Program Loading

- DBI Frameworks parse program binaries and create a code cache or hooks in order for further instrumentation to occur.

- Code cache is typically executed rather than original binary mapping.
Program Instrumentation

- Frameworks allow the registration of callbacks to handle events and insert instrumentation code.

- Callbacks are considered instrumentation routines and injected code are considered analysis routines.
Program Instrumentation

- Instrumentation hooks occur at varying granularity
  - Image Load
  - Trace
  - Function / Routine
  - Block
  - Instruction
Process Execution Events

- Callbacks for process execution events can be registered in addition to code loading events
  - Exceptions
  - Process attach
  - Process detach
  - Fini
  - Thread start
  - Thread exit
DBA Plugins

- Existing research has shown several uses for DBI frameworks
  - Diagnostic execution tracing
    - Call graph
    - Code coverage
    - Dataflow tracing
  - Heap profiling and validation
    - Think Application Verifier
  - Cache profiling
DBA Plugins

- Existing research has shown several uses for DBI frameworks
  - Mitigations
    - “Secure Execution Via Program Shepherding”
    - Control Flow Integrity

- Existing mitigations are not available or do not apply to modern Windows operating systems
Useless Benchmarks

- Benchmarking DBI frameworks is difficult

- The best benchmarks should measure CPU and memory efficiency against a shared analysis core

- We do not have this but lets look at some numbers anyway
Useless Benchmarks

C:\tools>yafu\yafu64
06/15/11 13:52:20 v1.20.2 @ BLACKHAWK, System/Build Info:
Using GMP-ECM 6.3, Powered by MPIR 2.1.1
detected Intel(R) Core(TM)2 Duo CPU T9900 @ 3.06GHz
detected L1 = 32768 bytes, L2 = 6291456 bytes, CL = 64 bytes
measured cpu frequency =~ 3035.702040

==================================================================
======= Welcome to YAFU (Yet Another Factoring Utility) =======
======= bbuhrow@gmail.com =======
======= Type help at any time, or quit to quit =======
==================================================================
cached 664581 primes. pmax = 10000079

<table>
<thead>
<tr>
<th>Fibonacci Sequence Benchmark</th>
<th>100000</th>
<th>250000</th>
<th>500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>1.42</td>
<td>7.379</td>
<td>28.143</td>
</tr>
<tr>
<td>DynamoRIO</td>
<td>1.607</td>
<td>7.472</td>
<td>28.891</td>
</tr>
<tr>
<td>PIN</td>
<td>2.402</td>
<td>8.377</td>
<td>29.219</td>
</tr>
</tbody>
</table>
Useless Benchmarks

C:\tools>ramspeed\ramspeed-win32.exe
RAMspeed (Win32) v1.1.1 by Rhett M. Hollander and Paul V. Bolotoff, 2002-09

USAGE: ramspeed-win32 -b ID [-g size] [-m size] [-1 runs]
-b runs a specified benchmark (by an ID number):
  1 -- INTmark [writing]
  2 -- INTmark [reading]
  3 -- INTmem
  4 -- FLOATmark [writing]
  5 -- FLOATmark [reading]
  6 -- FLOATmem
...

<table>
<thead>
<tr>
<th>Integer Benchmark (MB/sec)</th>
<th>Copy</th>
<th>Scale</th>
<th>Add</th>
<th>Triad</th>
<th>AVG</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>3451.85</td>
<td>3350.21</td>
<td>4022.76</td>
<td>3990.99</td>
<td>3703.95</td>
<td>23.182</td>
</tr>
<tr>
<td>DynamoRIO</td>
<td>3493.26</td>
<td>3335.9</td>
<td>3919.36</td>
<td>3839.93</td>
<td>3647.11</td>
<td>23.635</td>
</tr>
<tr>
<td>PIN</td>
<td>3382.53</td>
<td>3331.37</td>
<td>3767.52</td>
<td>3752.16</td>
<td>3558.39</td>
<td>24.633</td>
</tr>
</tbody>
</table>
Useful Benchmarks

- Benchmarks for security use are going to be highly subjective

- Criteria
  - Speed – Is the performance hit tolerable
  - Reliability – Does the tool limit false positives and not cause crashes on its own
  - Ease of Implementation – How long does it take to implement a tool under a particular DBI
RETURN ORIENTED PROGRAMMING
Return Oriented Programming

- Return Oriented Programming (ROP) is the modern term for “return-to-libc” method of shellcode execution

- ROP can be used to bypass DEP
  - VirtualProtect()
  - VirtualAlloc()
  - HeapCreate()
  - WriteProcessMemory()
Gadget Shellcode

- Gadgets are a series of assembly instructions ending in a return instruction

- Shellcode is executed by creating a fake call stack that will chain a series of instruction blocks together

```plaintext
## Generic Write-4 Gadget ##
rop += "\xD2\x9F\x10\x10"  # 0x10109FD2 :
    # POP EAX
    # RET
rop += "\xD0\x64\x03\x10"  # 0x100364D0 :
    # POP ECX
    # RET
rop += "\x33\x29\x0E\x10"  # 0x100E2933 :
    # MOV DWORD PTR DS:[ECX], EAX
    # RET
```
Gadget Shellcode

- Gadgets are a series of assembly instructions ending in a return instruction

- Shellcode is executed by creating a fake call stack that will chain a series of instruction blocks together

```
## Grab kernel32 pointer from the stack, place it in EAX ##
rop += "\x5D\x1C\x12\x10" * 6  # 0x10121C5D :
   # SUB EAX,30
   # RETN

rop += "\xF6\xBC\x11\x10"
   # 0x1011BCF6 :
   # MOV EAX, DWORD PTR DS:[EAX]
   # POP ESI
   # RETN

rop += rop_align
```
Gadget Shellcode

- Gadgets are a series of assembly instructions ending in a return instruction

- Shellcode is executed by creating a fake call stack that will chain a series of instruction blocks together

```
## EAX = kernel32 base, get pointer to VirtualProtect() ##
rop += ("\x76\xE5\x12\x10" + rop_align) * 4
    # 0x1012E576 :
    # ADD EAX,100
    # POP EBP
    # RETN

rop += "\x40\xD6\x12\x10"
    # 0x1012D640 :
    # ADD EAX,20
    # RETN

rop += "\xB1\xB6\x11\x10"
    # 0x1011B6B1 :
    # ADD EAX,0C
    # RETN

rop += "\xD0\x64\x03\x10"
    # 0x100364D0 :
    # ADD EAX,8
    # RETN

rop += "\x33\x29\x0E\x10"
    # 0x100E2933 :
    # DEC EAX
    # RETN

rop += "\x01\x2B\x0D\x10"
    # 0x100D2B01 :
    # MOV ECX,EAX
    # RETN

rop += "\xC8\x1B\x12\x10"
    # 0x10121BC8 :
    # MOV EAX,EDI
    # POP ESI
    # RETN
```
Gadget Shellcode

Small section of shellcode showing several gadgets chained together to locate kernel32!VirtualProtect()
Finding Gadgets

- Useful gadgets typically modify a pointer or cause a load or store operation
  - ADD, SUB, DEC, INC, DEC, PUSH, POP, XCHG, XOR

- Tools now exist for finding gadgets
  - msfpescan
  - Pvefindaddr – PyCommand for ImmunityDbg
Detecting ROP

- ROP requires the use of sub-sections of program blocks to create Gadgets

- Gadgets end in a RET instruction

- Normal program semantics generate call stacks that return to a code location immediately after a CALL or JMP instruction
Detecting ROP

- Algorithm

```plaintext
INSTRUMENT_PROGRAM
for each IMAGE
  for each BLOCK in IMAGE
    insert BLOCK in BLOCKLIST
  for each INSTRUCTION in BLOCK
    if INSTRUCTION is RETURN or BRANCH
      insert code to retrieve SAVED_EIP from stack
      insert CALL to ROP_VALIDATE(SAVED_EIP) before INSTRUCTION

ROP_VALIDATE
if SAVED_EIP not in BLOCKLIST
  exit with error warning
```
Detecting ROP

- Implementation
  - The initialization for our pintool is as simple as opening a log file and adding a couple hooks

```c
int main(int argc, char *argv[]) {
    PIN_InitSymbols();
    if (PIN_Init(argc, argv)) {
        return Usage();
    }

    outfile = fopen("c:\tools\antirop.txt", "w");
    if (!outfile) {
        LOG("Error opening log file\n");
        return 1;
    }

    PIN_AddFiniFunction(Fini, 0);
    TRACE_AddInstrumentFunction(Trace, 0);

    LOG("[+] AntiROP instrumentation hooks installed\n");
    PIN_StartProgram();
    return 0;
}
```
Detecting ROP

- Implementation
  - This function implements the callback function when PIN loads a trace of basic blocks the first time and instruments RET instructions.

```c
VOID Trace(TRACE trace, VOID *v)
{
    ADDRINT addr = TRACE_Address(trace);

    // Visit every basic block in the trace
    for (BBL bbl = TRACE_BblHead(trace);
         BBL_Valid(bbl);
         bbl = BBL_Next(bbl))
    {
        for(INS ins = BBL_InsHead(bbl);
            INS_Valid(ins);
            ins=INS_Next(ins))
        {
            ADDRINT va = INS_Address(ins);
            if(INS_IsBranchOrCall(ins))
            {
                Calls.insert(va);
            }
            if(INS_IsRet(ins))
            {
                INS_InsertCall(ins,
                                IPOINT_BEFORE,
                                AFUNPTR(AntiROPRetCheck),
                                IARG_INST_PTR,
                                IARG_REG_VALUE,
                                REG_STACK_PTR,
                                IARG_END);
            }
        }
    }
}
```
Detecting ROP

- Implementation
  - This function executes before every RET or indirect branch is executed to validate the saved return value points to an instruction after a call.

```c
VOID AntiROPRetCheck(ADDRINT va, ADDRINT esp)
{
    UINT32 *ptr = (UINT32 *)esp;
    for(int i = 0; i < 4; i++)
    {
        if(* (ptr + i) == 0x90909090)
        {
            fprintf(outfile,
                  "NOPS FOUND AT ESP + %d: [%x] = 0x90909090\n",
                    i, ptr + i);
        }
    }
    CallsIter = Calls.find(*ptr);
    if (CallsIter != Calls.end())
    {
        count = 0;
    }
    else if(++count > threshold)
    {
        ReportAntiROP(*ptr, count, threshold);
    }
    fflush(outfile);
}
```
Detecting ROP

● Output

```
C:\tools>pin\pin.bat -t mypintool.dll -AntiROPRet -- kmplayer\KMPlayer.exe
C:\tools>type antirop.txt
NOPS FOUND AT ESP + 1: [1196b5f4] = 0x90909090
NOPS FOUND AT ESP + 2: [1196b5f8] = 0x90909090
NOPS FOUND AT ESP + 3: [1196b5fc] = 0x90909090
ANTI-ROP detected an attempted RET to 100ebf17 without using a CALL .. exiting
```
Other Mitigations

- ROPDefender
  - Shadow stack
    - Hook before CALL to store return address
    - Hook before RET to determine if returning to address stored before CALL

- SHAN
  - Branch monitoring
    - Store each valid basic block in a list before execution
    - At runtime verify branch destination is in list
JUST-IN-TIME SHELLCODE
Just-In-Time Shellcode

- Just-in-Time (JIT) Shellcode is emitted by a JIT compiler while converting bytecode of an interpreted language to native machine code.

- Scripting code such as ActionScript or Javascript is supplied by the user and therefore creates potential for control of native code in the process address space.
Just-In-Time Shellcode

- The JIT process creates a writable and executable page with user controlled data.

- If an attacker can manipulate the emitted machine code, it can be used to the advantage of the attacker to bypass mitigations.
Just-In-Time Shellcode

- Published research has shown that using math operators, specifically XOR, leads to controllable machine code output

<table>
<thead>
<tr>
<th>Operator ADD (+):</th>
<th>Operator XOR (^):</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ b8 90 90 90 3c ] mov eax,03 c909090h</td>
<td>[ b8 90 90 90 3c ] mov eax, 3c909090h</td>
</tr>
<tr>
<td>[ f2 0f 2a c0 ] cvtsi2sd xmm0, eax</td>
<td>[ 35 90 90 90 3c ] xor eax, 3c909090h</td>
</tr>
<tr>
<td>[ 66 0f 28 c8 ] movapd xmm1, xmm0</td>
<td>[ 35 90 90 90 3c ] xor eax, 3c909090h</td>
</tr>
<tr>
<td>[ f2 0f 58 c8 ] addsd xmm1, xmm0</td>
<td>[ f2 0f 58 c8 ] addsd xmm1, xmm0</td>
</tr>
<tr>
<td>[ f2 0f 58 c8 ] addsd xmm1, xmm0</td>
<td></td>
</tr>
</tbody>
</table>
Just-In-Time Shellcode

Published research has shown that using math operators, specifically XOR, leads to controllable machine code output

```
var y=(0x11223344^0x44332211^0x44332211...);
```

Compiles as:

```
0x909090: 35 44 33 22 11 XOR EAX, 11223344
0x909095: 35 44 33 22 11 XOR EAX, 11223344
0x90909A: 35 44 33 22 11 XOR EAX, 11223344
```
Just-In-Time Shellcode

- Published research has shown that using math operators, specifically XOR, leads to controllable machine code output

Disassemble at a byte offset to get useful code:

0x909091: 44          INC ESP
0x909092: 33 22       XOR ESP, [EDX]
0x909094: 11 35 44 33 22 11  ADC [11223344], ESI
0x90909A: 35 44 33 22 11  XOR EAX, 11223344
Just-In-Time Shellcode

- The native behavior of the JIT compiler results in an automatic DEP bypass

- Once a usable payload is constructed using specialized arguments around the XOR operator the executable payload must be found

- Heapspray or memory leak
  - See Dion Blazakis’s paper “Interpreter Exploitation”
Detecting JIT Shellcode

● The ActionScript and JavaScript JIT compilers change memory permissions of compiled machine code to R-E rather than RWE before execution

● We have seen that currently known JIT shellcode relies heavily on the XOR operator
Detecting JIT Shellcode

- We can use a simple heuristic by hooking `kernel32!VirtualProtect` and checking the disassembly for an unusual number of XORs.

- Piotr Bania also pointed out a primitive that can be used to identify operators:

```
mov     reg, IMM32
operation reg, IMM32
operation reg, IMM32
operation reg, IMM32

...```

Detecting JIT Shellcode

- **Algorithm**

**INSTRUMENT_PROGRAM**
Insert CALL to JIT_VALIDATE at prologue to VirtualProtect

**JIT_VALIDATE**
Disassemble BUFFER passed to VirtualProtect for each INSTRUCTION
  - if INSTRUCTION is MOV_REG_IMM32 then
    - while NEXT_INSTRUCTION uses IMM32
      - increase COUNT
    - if COUNT > THRESHOLD then
      - exit with error warning
Detecting JIT Shellcode

- Implementation
  - The initialization for our pintool is as simple as opening a log file and adding a couple hooks

```c
int main(int argc, char *argv[]) {
  PIN_InitSymbols();
  if(PIN_Init(argc, argv)) {
    return Usage();
  }

  outfile = fopen("c:\tools\antijit.txt", "w");
  if(!outfile) {
    LOG("Error opening log file\n");
    return 1;
  }

  IMG_AddInstrumentFunction(ModuleLoad, NULL);

  LOG("[+] AntiJIT instrumentation hooks installed\n");

  PIN_StartProgram();
  return 0;
}
```
Detecting JIT Shellcode

- Implementation
  - This function implements the callback function when PIN loads a module so that VirtualProtect may be hooked

```c
void ModuleLoad(IMG img, VOID *v)
{
    RTN rtn;
    
    rtn = RTN_FindByName(img, "VirtualProtect");
    if (RTN_Valid(rtn))
    {
        RTN_Open(rtn);
        
        RTN_InsertCall(rtn,
                       IPOINT_BEFORE,  
                       AFUNPTR(VirtualProtectHook),
                       IARG_FUNCARG_ENTRYPOINT_VALUE, 0,  // lpAddress
                       IARG_FUNCARG_ENTRYPOINT_VALUE, 1,  // dwSize
                       IARG_END);
        
        RTN_Close(rtn);
    }
}
```
Detecting JIT Shellcode

- Implementation
  - This function executes before calls to VirtualProtect to disassemble the target buffer and determine if a JIT shellcode is probable

```c
void VirtualProtectHook(VOID *address, SIZE_T dwSize)
{
    // Disassemble buffer into linked list
    ...
    while(insn && !MOV_IMM32(insn))
    {
        insn = insn->next;
        while(insn)
        {
            if(OP_IMM32(insn))
                count++;
            if(count > threshold)
                ReportAntiJIT();
            insn = insn->next;
        }
    }
}
```
QUESTIONS
Q & A

- VRT information:
  - Web – http://www.snort.org/vrt
  - Blog – http://VRT-sourcefire.blogspot.com/
  - Twitter – @VRT_sourcefire
  - Videos – http://vimeo.com/vrt
  - Labs – http://labs.snort.org

Richard Johnson
rjohnson@sourcefire.com
rjohnson@uninformed.org
@richinseattle
References