ROP gadgets in Metasploit

Andrew Ruef (andrew@trailofbits.com)

Introduction

Return Oriented Programming(ROP) is a common technique used in the creation of code execution vulnerabilities. As Metasploit contains many implementations of exploits in native code, it also has many representative samples of ROP payloads used in exploits. Metasploit represents the collective efforts of hundreds of computer security experts; it probably represents the single largest (open source and free) compendium of what exploit development is possible.

To determine the types of programs exploit authors write using ROP, we surveyed the collection of Windows browser exploits that use ROP and characterized the types of programs written and the sequences used.

Characterizations

After survey, there are two kinds of ROP gadget collections in Metasploit. There are gadgets assembled using a tool known as 'mona'. Mona is an extension of the Immunity Debugger, it performs analysis on program states to identify a sequence of instructions that can be formed into a gadget. This gadget will set the page permissions of a region of memory to executable, and then branch to it.

Case 1: Adobe Flash Player MP4 'cprt' Overflow

This exploit used mona to identify sequences that result in a VirtualProtect to enable executable pages on memory. The most complicated sequence used took the following form:

```
mov eax, [eax]
pop ebp
ret
```

The following transfer statements can easily characterize this sequence:

```
EAX = MEMREAD[EAX]
EBP = MEMREAD[ESP]
EIP = MEMREAD[Add[ESP, 4]]
```

Here we can see that we would need to adjust the stack pointer to preserve the semantics, as the memory read would take place after the 'pop'.

Case 2: IBM Tivoli Provisioning Manager Express for Software Distribution Isig.isigCtl.1 ActiveX RunAndUploadFile() Method Overflow

This exploit also used mona to identify sequences. This used a complicated sequence that would restore many registers from the stack. The sequence in assembly code is as follows:

```
pop eax
pop edi
pop esi
pop ebx
pop ebp
ret
```

We can model this using transfer statements, as follows:

```
EAX = MEMREAD[ESP]
EDI = MEMREAD[Add[ESP, 4]]
ESI = MEMREAD[Add[ESP, 8]]
EBX = MEMREAD[Add[ESP, 12]]
EBP = MEMREAD[Add[ESP, 16]]
EIP = MEMREAD[Add[ESP, 20]]
```

Case 3: MS10-002 Internet Explorer Object Memory Use-After-Free

This exploit uses mona as well. It contains a few different types of sequences to effect different actions. One of these sequences is the 'pushad' instruction. This instruction is interesting because on its face it seems to be a single statement, but when it is expanded into a semantics-representing IR it is revealed that the instruction actually preserves all the current general purpose registers onto the stack in a specific order. So the native code of the form:

```
pushad
ret
```

Is specified in transfer statements with the following:

```
MEMWRITE[ Add[ESP, 28 ]] = EAX
MEMWRITE[ Add[ESP, 24 ]] = ECX
MEMWRITE[ Add[ESP, 20 ]] = EDX
MEMWRITE[ Add[ESP, 16 ]] = EBX
MEMWRITE[ Add[ESP, 12 ]] = ESP
MEMWRITE[ Add[ESP, 8 ]] = EBP
MEMWRITE[ Add[ESP, 4 ]] = ESI
MEMWRITE[ESP] = EDI
EIP = MEMREAD[ Add[ESP, 32] ]
```

Case 4: VLC AMV Dangling Pointer Vulnerability

This exploit uses mona. It uses sequences that we have previously discussed, as well as one of the following form:

```
add ebx, eax
xor eax, eax
inc eax
ret
```

We can represent this sequence using transfer statements of the following form:

```
EAX = 1
EAX = Add[EAX, EBX]
EIP = MEMREAD[ESP]
```

Case 5: Sun Java Runtime New Plugin docbase Buffer Overflow

This exploit does not use mona, rather, it is apparently hand crafted. It makes use of instructions and sequences that the other exploits do not in their gadgets, but many of them are the same. The one that differs is this sequence:

```
rep movsd
pop edi
pop esi
sub eax, eax
ret
```

Representing this sequence with a transfer statement will be discussed later.

Similarities and differences

The case 5 uses an instruction that describes a loop, the rep-prefixed movs instruction. This instruction is different from other statements in assembler language in two ways. The first is that it contains a memory to memory semantic, in that both the ‘esi’ and ‘edi’ registers are treated as pointers to memory and dereferenced by the semantics of the instruction. The second is that when this instruction is prefixed with ‘rep’, it will conditionally loop until the ‘ecx’ register is 0, and decrement the ‘ecx’ register. When the semantics of ‘rep movs’ are represented in the IL, they include a back-edge creating a loop.

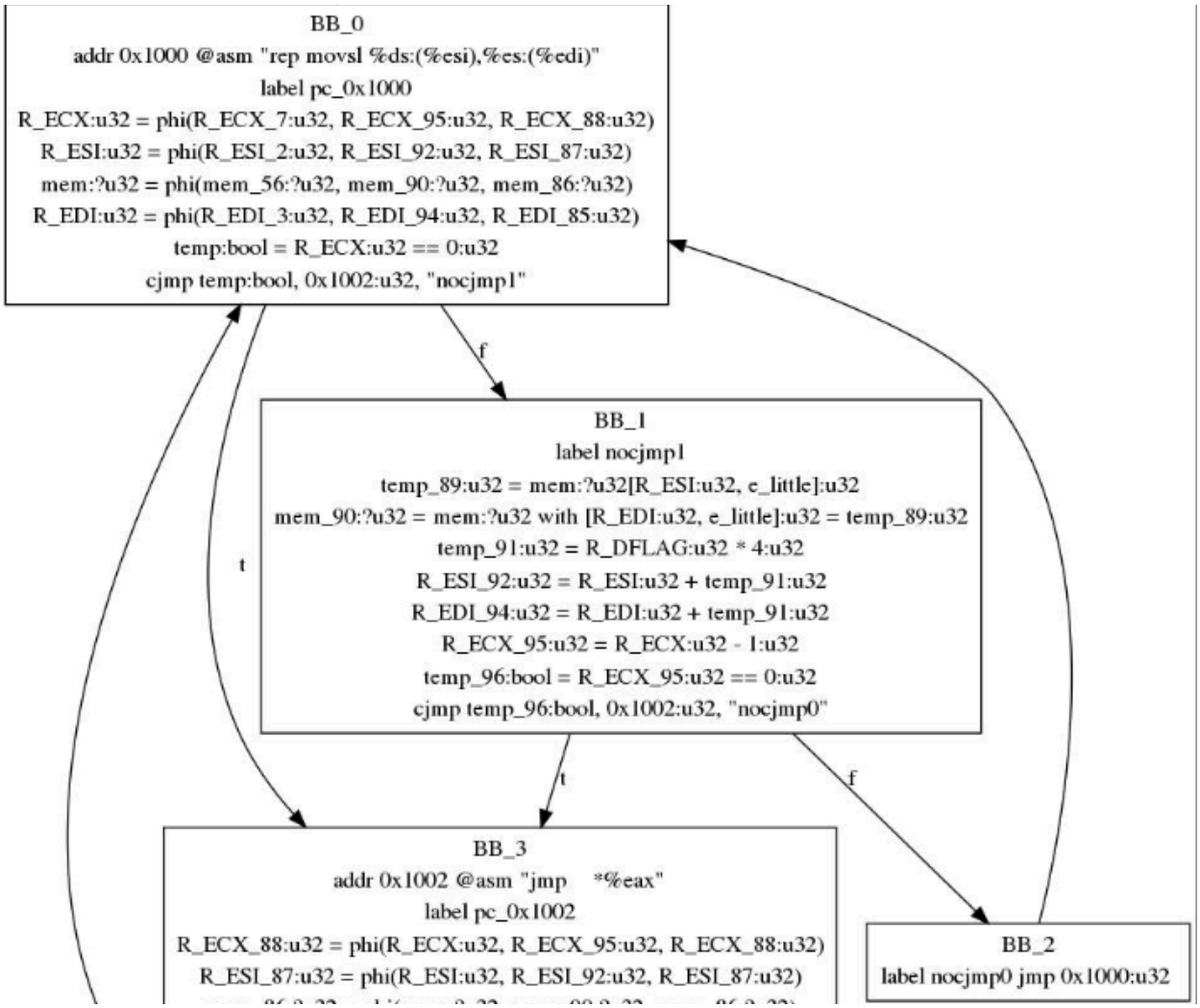

Figure generated using BAP

This is interesting because automated reasoning about this kind of behavior and its summarization is difficult. The existing automated tools (mona) shy away from this and use sequences with clear pre and post conditions. An expert, working by hand, can use a sequence that has complicated behavior.

Generalizations

Our language allows us to generalize specifications somewhat. In the case of register writes from stack, we can create expressions that state “reads some value from the stack into any register”. We can also provide general definitions of sequences such as the above ‘rep movsd’. The loop expresses a data copy operation, and the semantics of our data copy operation can be represented using an expansion of transfer statements.

We can think of this intuitively as unrolling a loop. For example, a statement of the form:

```
for( int i = 0; i < K; i++) {
       dst_buf[i] = src_buf[k];
```

```
}
```

This statement can be unrolled as long as we know the bounds of K. So in our language, we would create a statement that looked something like this:

```
x0 = _
MEMWRITE[x0] = MEMREAD[EAX]
MEMWRITE[Add[x0, 4]] = MEMREAD[Add[EAX, 4]]
MEMWRITE[Add[x0, 8]] = MEMREAD[Add[EAX, 8]]
…
```

In this way we can search for regions of code that either contain branching behavior, as our searching logic can be made to summarize loops, or we can find regions of code that work without loops. For example, if there is a data copy in code that is a flattened loop, we can take advantage of the semantics of the flattened block to effect data transfer across long strides in memory.

We can also use our notion of temporary values to state that for each MEMWRITE statement, the 'base' of the write can be any register, but it must be the same register across all of the writes.

Conclusion

After a survey of the Metasploit software library, we believe that it's possible to represent modern ROP-based exploits using our language and we have many examples cataloged. We also believe that it's possible to represent far more than what is currently in Metasploit, which implies that our language is more general and this solution is future-proof.