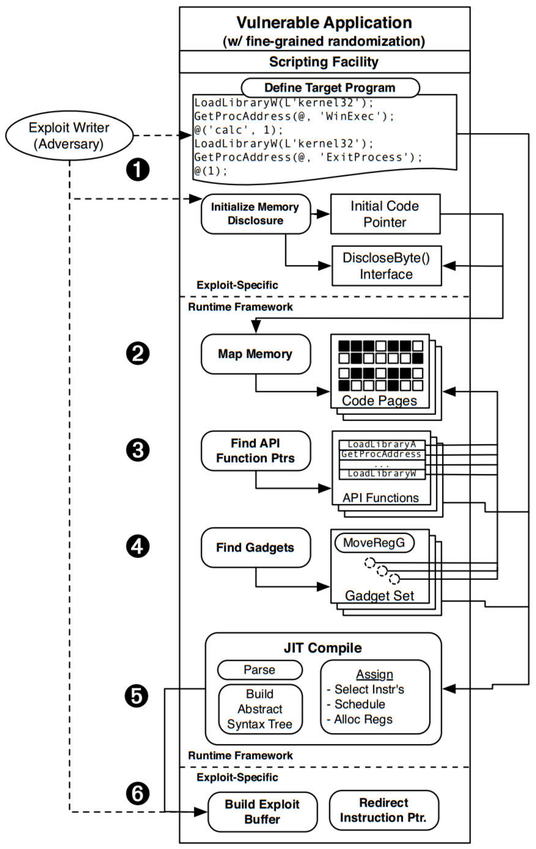
Chobham: Taming JIT-ROP Attacks

Ben Niu & Gang Tan
Lehigh University

What are JIT-ROP Attacks?

JIT-ROP, or Just-In-Time Return-Oriented Programming, is an attack form that defeats fine-grained ASLR. It harvests ROP gadgets at runtime by exploiting memory disclosure bugs and reading code pages. Then it compiles the attack payload using the gadgets found. The workflow is shown in the following diagram.



It should be noted that JIT-ROP attacks do not necessarily require a JIT compiler. Any exploit that finds and chains ROP gadgets on-the-fly can be considered as JIT-ROP. Since JIT-ROP requires (1) ***memory disclosure***; and (2) ***ROP gadget chaining***, we can mitigate it by restricting disclosed memory and raising the bar of chaining ROP gadgets, shown on the right.

CFI with Input-triggered CFG Generation

Control-Flow Integrity (CFI) is a general approach to defending against ROP attacks. Traditionally, CFI precomputes a CFG for the victim program, and instruments the program's binary to ensure that any runtime control-flow should not deviate from the CFG. CFI, especially when a fine-grained CFG is generated for the target program, makes it more difficult to chain ROP gadgets, since the chain has to follow a control-flow path allowed by the benign program. Thus in general, the finer the CFG is, the more security we gain from CFI in defeating ROP attacks. Although it is possible that we can further improve our static analysis to extract more precise CFGs, we still have a problem that the CFG has to be "large" enough to cover all possible program inputs. Therefore, even in the "perfect" CFG, there might still be lots of functionality-irrelevant control-flow edges that are not needed for a concrete input. These edges might be ammunition for attackers.

Instead, we propose input-triggered CFI (ITCFI), which generates the CFG required for each concrete input at runtime. Observing that any program has to firstly define a target address then use it in an indirect control-flow transfer, we can dynamically add edges to the CFG after a target is firstly defined. The following indirect branch targets need to be considered:

- *Virtual methods*. Only if a C++ class's constructor is invoked, then all its virtual methods' be reachable.
- *Global functions or static member methods*. Only if their addresses are explicitly taken can they be reachable.
- *Return addresses*. Only if a return address's preceding call instruction is executed can it be reachable.
- *Catch clauses*. Only if a catch clause whose function has been entered for the first time can it be reachable.

For each kind of the above, we can instrument its definition site to add it as reachable in the CFG. In addition, we can provide APIs for developers to dynamically *drop* CFG edges.

The performance, compared to the conventional CFI, would be worse due to dynamic CFG edge addition. However, since the edges are added once, we can dynamically patch the edge addition instrumentation code to minimize the overhead to a per-program constant. For example, a return address can be added into the CFG in the following way:

Compiled code	**Load-time code patch**	**Run-time code patch**
call printf	call CFG_Add_RA	call printf
ra_printf:	ra_printf:	ra_printf:

Callee-saved Register Restoration Randomization

Conventionally, attackers can use call-preceded gadgets to control register values. However, CFI makes it hard for attackers to control arbitrary register values, but still easy for those callee-saved registers defined by the calling convention. During attacking, the attackers need to control such register values and find a control-flow path to pass these values to function argument passing registers (on x86-64). By randomizing the order in which the registers are restored, we can raise the bar of attacking.

```
call foo                          foo:
movq %rbx, %rdi                   // omitted
movq %rbp, %rsi                   pop  %rbx
movq %r12, %rdx                   pop %rbp
call mprotect                     pop %r12
                                  ret
```

Load-time randomization

```
foo:
// omitted
pop  %rbp   // Functions'  prologues as well
pop %r12    // as stack unwinding data (e.g., eh_frame)
pop %rbx    // also need to be changed.
ret
```

Heap Zone

For programs, especially web browsers that heavily use the memory heap, we could identify the objects that are always in the heap and sanitize their methods to check whether the access happens in memory areas other than the heap.

For each program, we allocate a heap zone in its address space that only holds program-allocated heap objects. No stack and code would be in the heap zone, therefore, any heap buffer overflow (sequential) or UAF will not directly steal or pollute information in the code and stack. For objects (e.g., JavaScript-accessible objects) that only operate on the heap, we can sanitize their access methods to make sure their access only happens in the Heap Zone.

For example, we can add checks to the ArrayBuffer object to make sure its element read and write only happen in the heap instead of the code or stack..