Attacking Client Side JIT Compilers

BlackHat 2011

matasano SECURITY

Introduction

Chris Rohlf - Principal Security Consultant

@chrisrohlf
chris@matasano.com

Yan Ivnitskiy - Security Consultant

@yan
yan@matasano.com

http://www.matasano.com/research

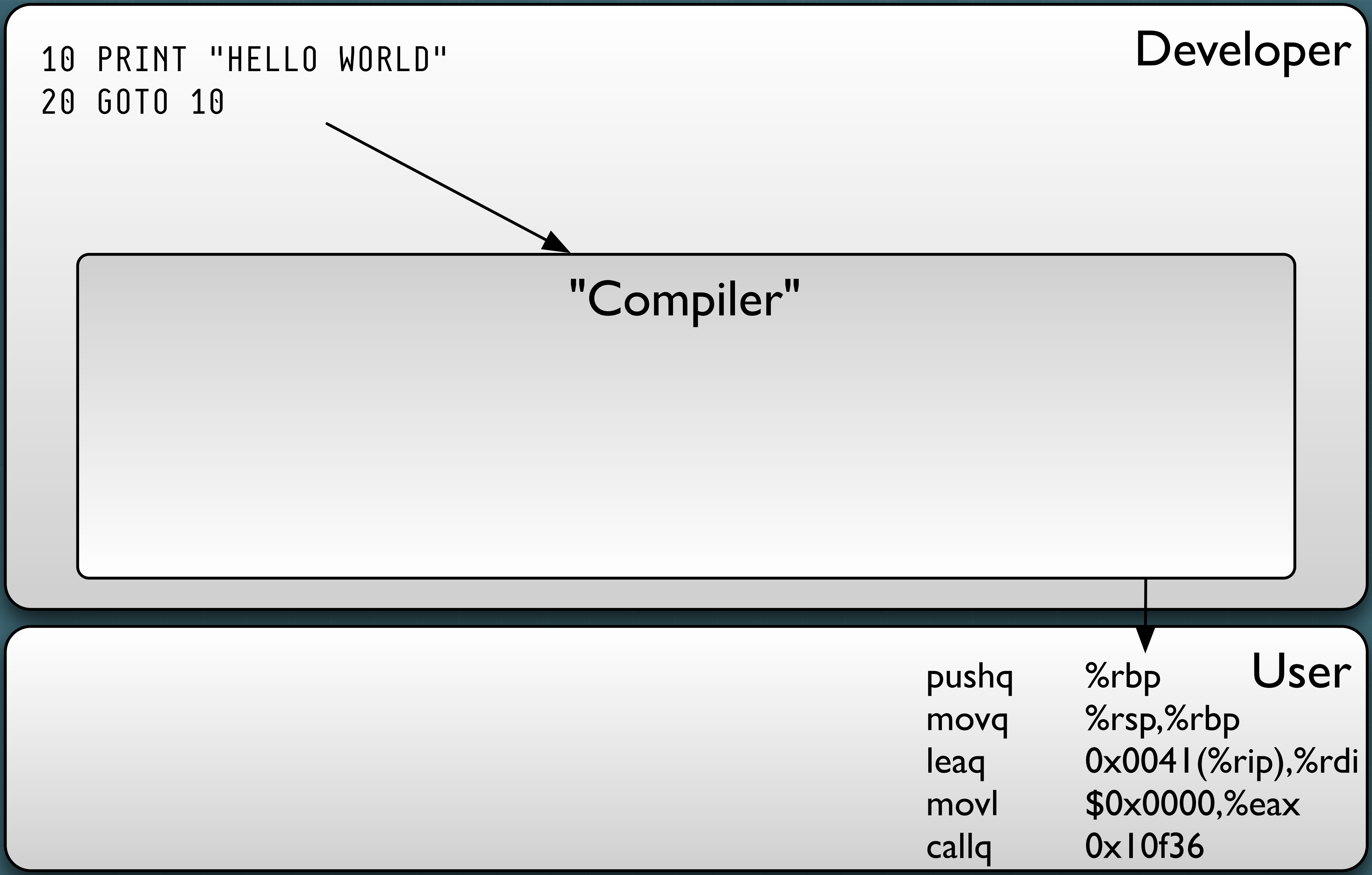
Overview

- Introduction
- Firefox JIT(s)
- LLVM JIT
- JIT Code Emission Bugs
- JIT Exploitation Primitives
- JIT Hardening
- JIT Engine Comparison
- Our Tools and Techniques

Introduction to JITs

- Interpreters and JIT Engines
  - Parse high level languages
  - Generate bytecode
  - Optimize and compile bytecode to native code
- They are everywhere
  - Browsers
  - Language runtimes (Java, Ruby, C#)

Introduction to JITs



Developer

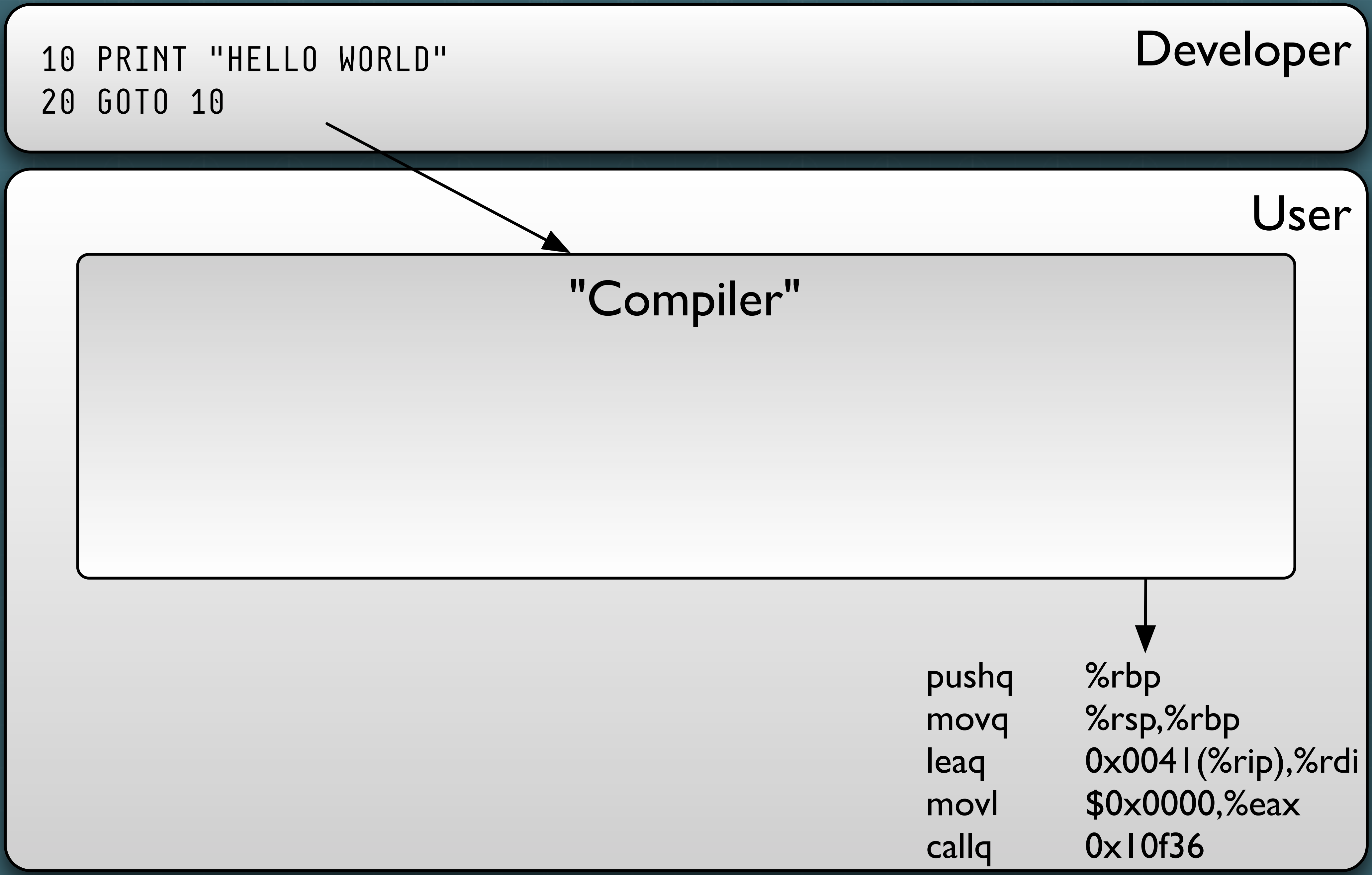
```
10 PRINT "HELLO WORLD"
20 GOTO 10
```

"Compiler"

User

```
pushq     %rbp
movq      %rsp,%rbp
leaq      0x0041(%rip),%rdi
movl      $0x0000,%eax
callq     0x10f36
```

Introduction to JITs



Developer

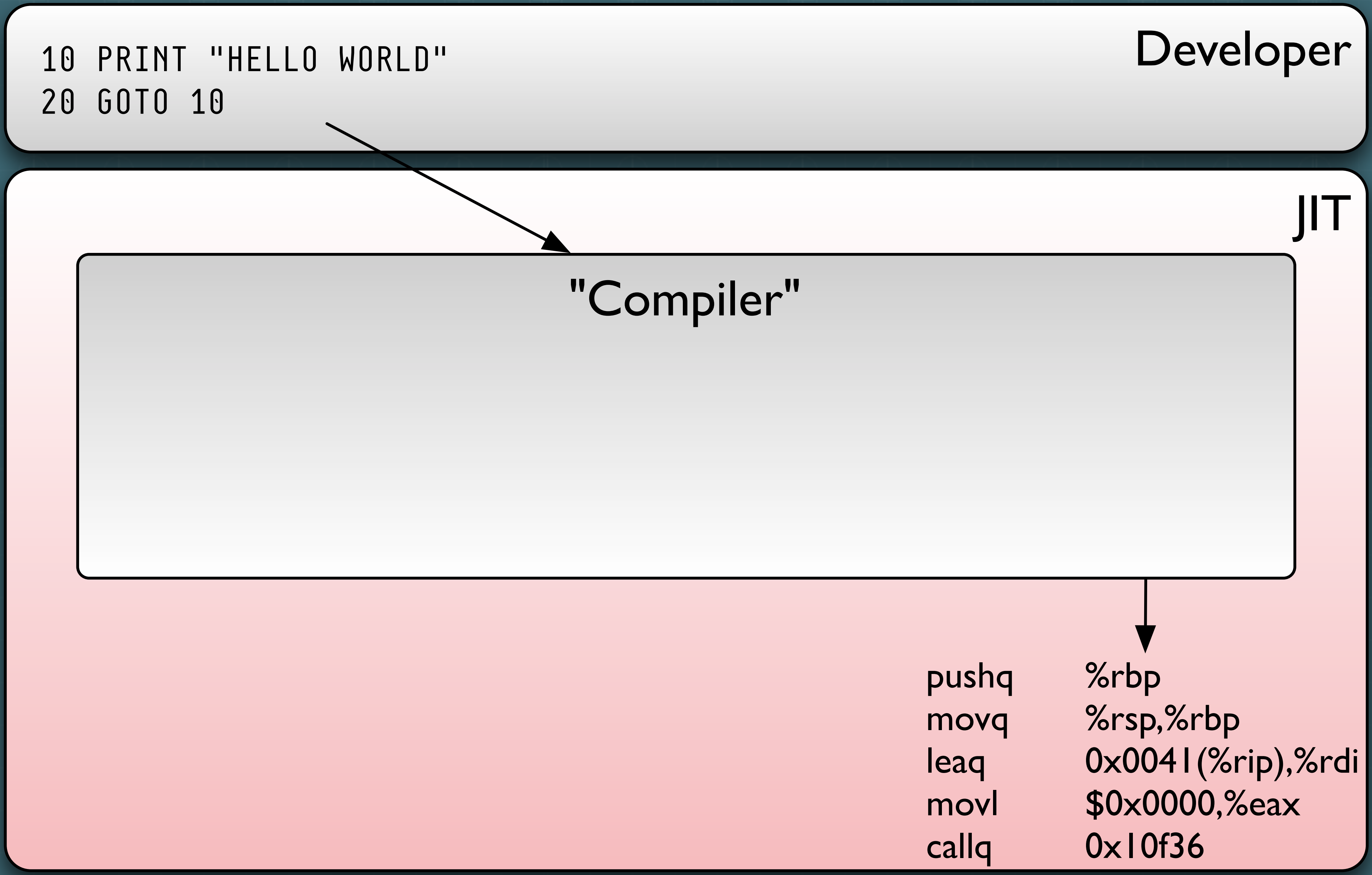
```
10 PRINT "HELLO WORLD"
20 GOTO 10
```

User

"Compiler"

```
pushq     %rbp
movq      %rsp,%rbp
leaq      0x0041(%rip),%rdi
movl      $0x0000,%eax
callq     0x10f36
```

Introduction to JITs



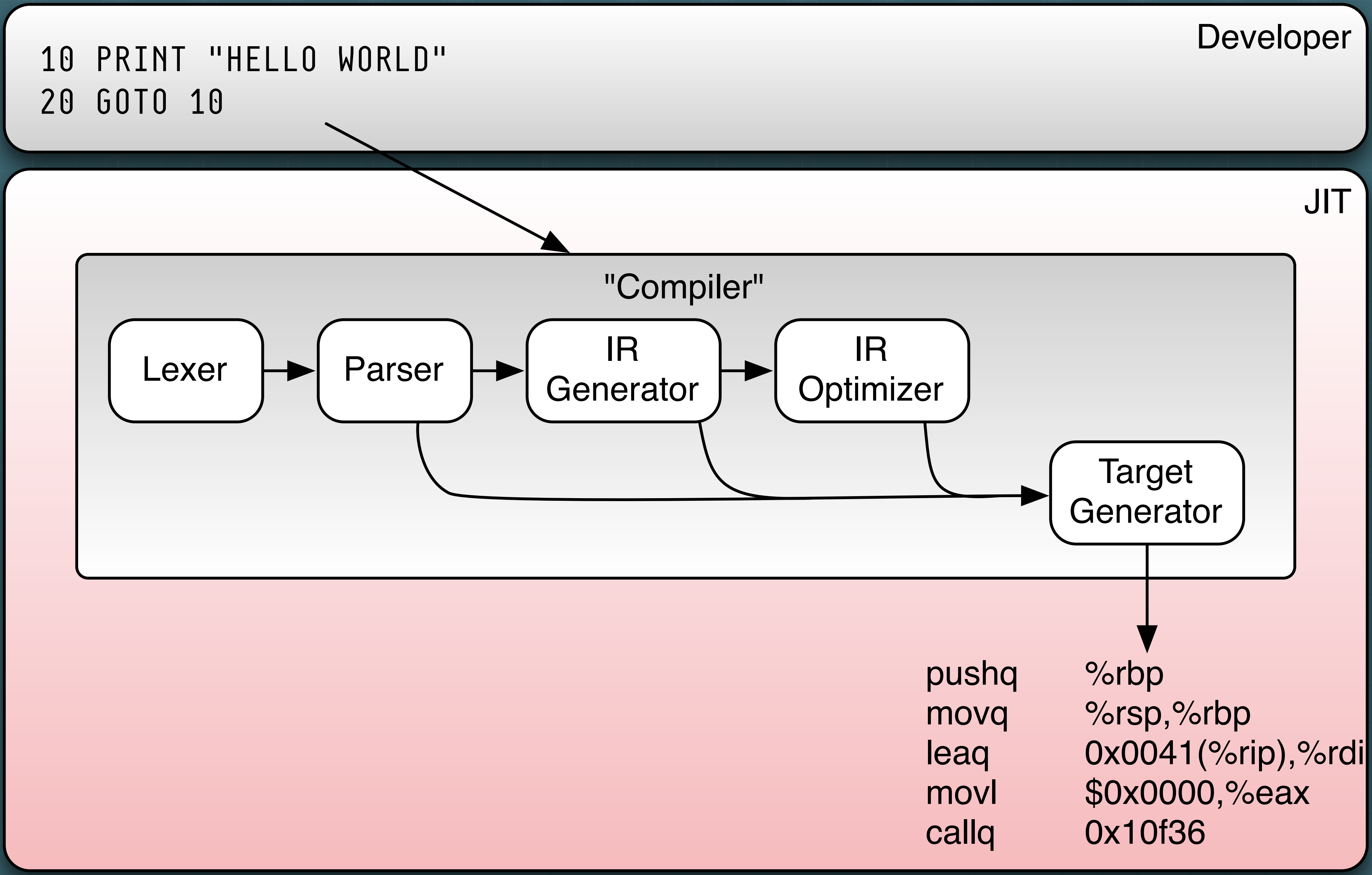
```
10 PRINT "HELLO WORLD"
20 GOTO 10
```

```
pushq    %rbp
movq     %rsp,%rbp
leaq     0x0041(%rip),%rdi
movl     $0x0000,%eax
callq    0x10f36
```

Introduction to JITs

- Compilers and JITs have been around for a while and come in a few different designs and architectures

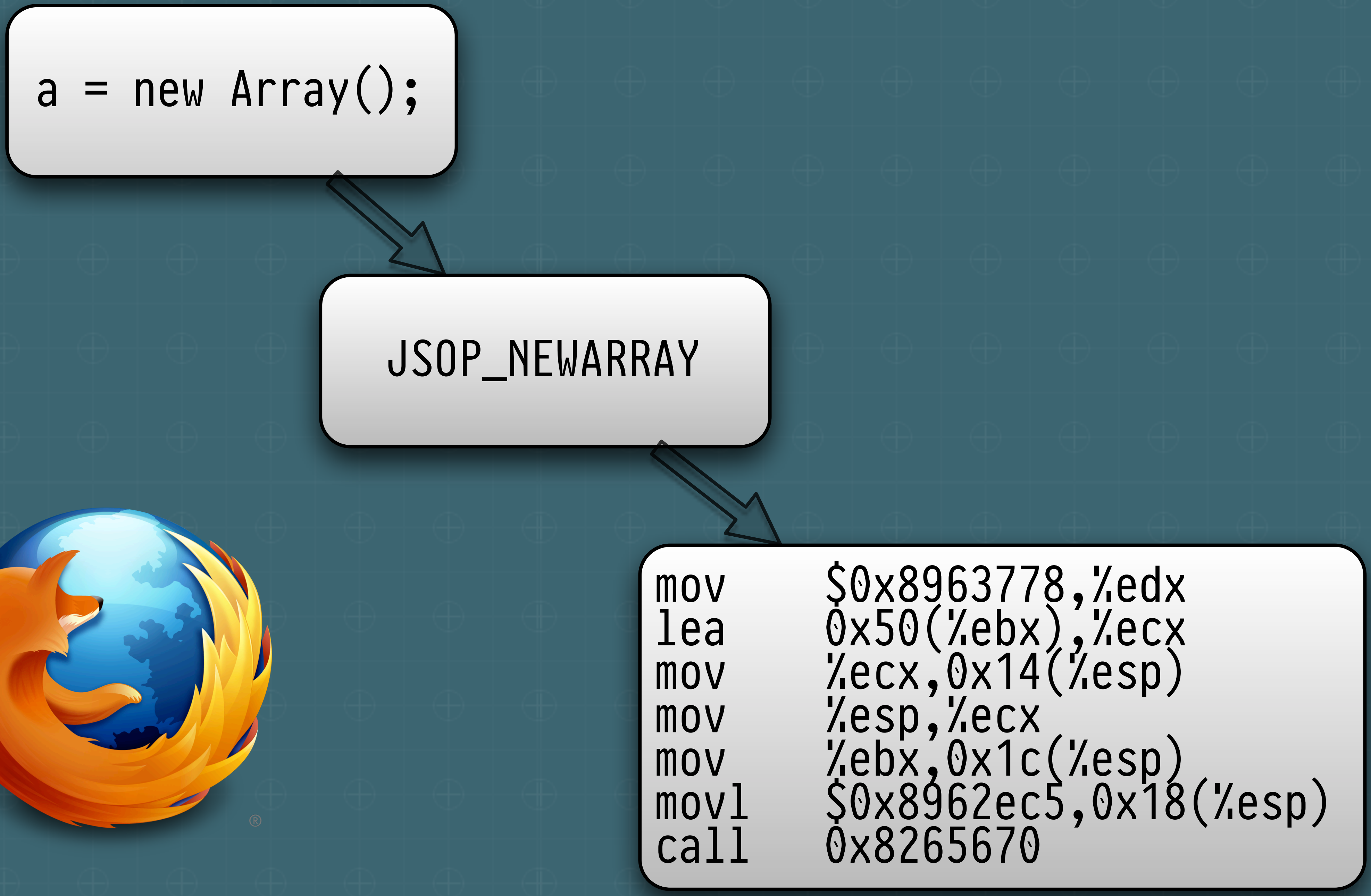
Introduction to JITs



```
10 PRINT "HELLO WORLD"
20 GOTO 10
```

```
pushq     %rbp
movq      %rsp,%rbp
leaq      0x0041(%rip),%rdi
movl      $0x0000,%eax
callq     0x10f36
```

Introduction to JITs



```
a = new Array();
```

```
JSOP_NEWARRAY
```

```
mov     $0x8963778,%edx
lea     0x50(%ebx),%ecx
mov     %ecx,0x14(%esp)
mov     %esp,%ecx
mov     %ebx,0x1c(%esp)
movl    $0x8962ec5,0x18(%esp)
call    0x8265670
```

Introduction to JITs

- Bytecode / Bitcode / Intermediate Representation (IR)
  - Both trusted and untrusted
  - Expressive and bloated (slower)
  - Simple and slim (faster)
  - Potentially usable to an attacker
    - Overwrite bytecode

Introduction to JITs

- Untrusted bytecode
  - Can be delivered from untrusted sources
    - Flash, CLR, LLVM
  - Completely external to the compiler
- Trusted bytecode
  - Produced internally by a trusted front end
    - SpiderMonkey
  - Still potentially usable to an attacker with control of the process

Introduction to JITs

- Tracing
  - Only JIT CPU-intensive code
  - Enables Optimizations
  - Types are generally known from tracing

TraceMonkey

- Introduced in Firefox 3.5
- Tracing JIT
- Uses NanoJIT as a backend assembler

matasano SECURITY

TraceMonkey

- TraceMonkey JITs hot code blocks
  - The recorder traces execution of SpiderMonkey IR
    - 8 Iterations before TraceMonkey kicks in
  - Produces trace trees
  - Emits optimized LIR for NanoJIT to compile
- Doesn’t handle type changes well

TraceMonkey

- CodeAlloc class
  - Handles allocating JIT pages that will hold code
  - Allocates memory RWX
- CodeList class
  - Inline meta-data for tracking the location of code chunks within JIT pages
  - *Next *Lower *Terminator pointers at static offsets
    - Creates a doubly linked list of JIT pages
    - Overwriting these will give you an arbitrary 4 byte write
    - Similar to the original heap unlink attacks

Introduction to JITs

- Method
  - JITs entire functions / methods
  - Usually generates unoptimized code
    - Not based on previous execution runs
  - Slow type lookups are usually required

JaegerMonkey

- Introduced in Firefox 4.0
- Method JIT
- Uses the Nitro assembler backend from WebKit
- SpiderMonkey bytecode → Native Code
- Uses an Inline Cache for handling type changes in property accesses

JaegerMonkey

- Fast paths are native code emitted by the JIT
  - Pure native code emitted by the JIT for predefined operations
- Slow paths are through the execution of bytecode
  - Inline cache hits sometimes have to go back through slow bytecode execution
- Stub calls are into C++ code from JIT pages
  - Typically exist to augment a fast path

JaegerMonkey

- ExecutableAllocator class
  - Handles allocating JIT pages to hold code
  - Allocates memory RWX
- ExecutablePool class
  - Handles managing the larger page size allocations into pools to hold native code
  - Pools are chosen based on the size of code that needs to be stored

Inline Caching

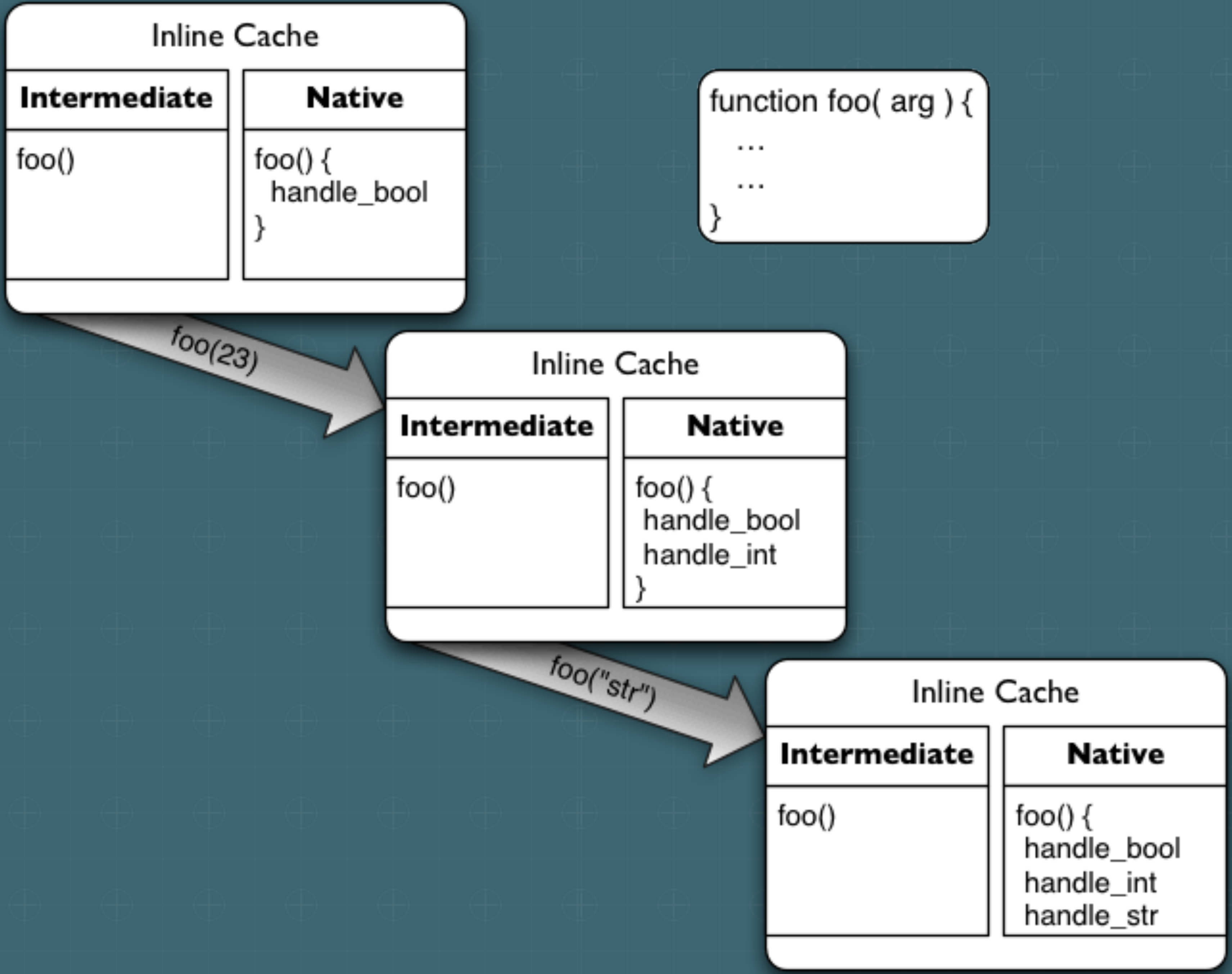
- Inline Caching
  - JavaScript is dynamically typed
  - How do you JIT a generic function that handles multiple types?

```
function a = blah(var b) {
   for(i=0; i<10; i++) {
      b += i;
   }
}

blah(“hello”);
blah([0, 1, 2, 3]);
```

  - Inline caches handle rewriting methods or property accesses at runtime to handle different and unexpected types

Inline Caching



Inline Cache

Intermediate	Native

foo()	foo() {
	 handle_bool
	}

```
function foo( arg ) {
   …
   …
}
```

foo(23)

Inline Cache

Intermediate	Native

foo()	foo() {
	 handle_bool
	 handle_int
	}

foo("str")

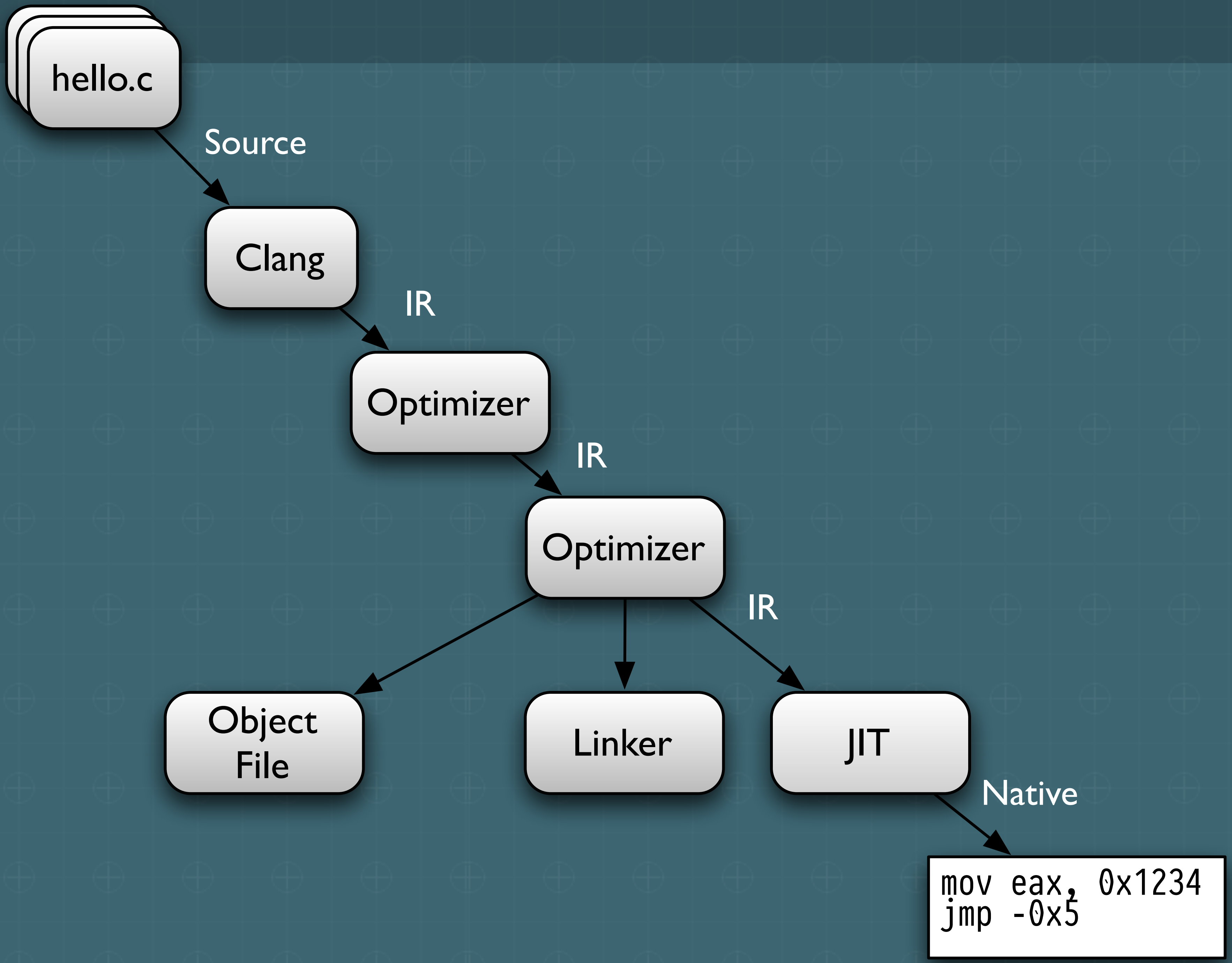
Inline Cache

Intermediate	Native
foo()	foo() { handle_bool handle_int handle_str

LLVM

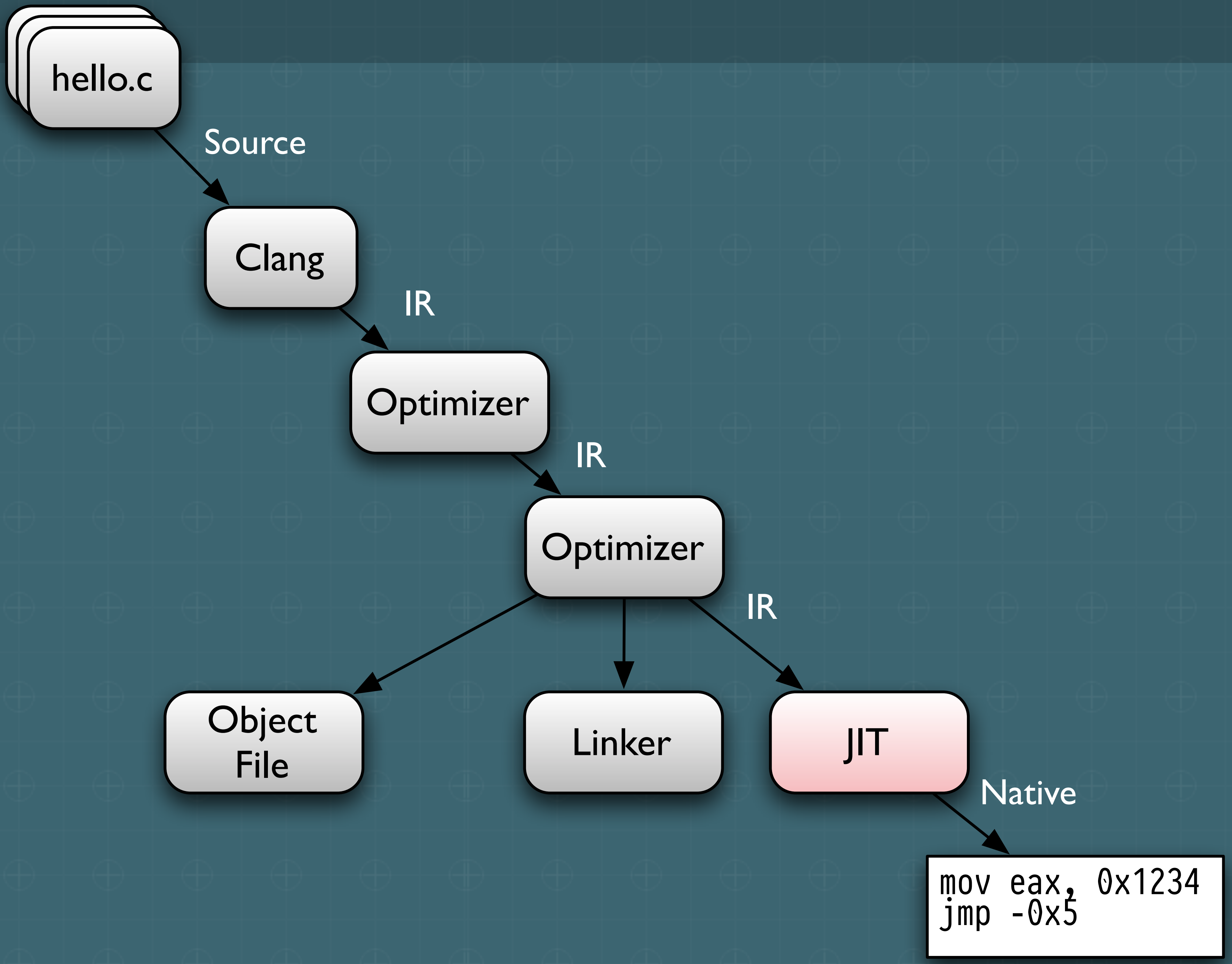
- An instruction set, a suite of libraries and a collection of tools designed around compilation.
- A suite of libraries from the start
- Initially used GCC as a front end
- Now supports C, C++ and Objective-C natively
- Many other compiler projects now support LLVM
  - Python, Ruby, Haskell, PHP, etc
- Popular for implementing compiler back ends

LLVM



```
mov eax, 0x1234
jmp -0x5
```

LLVM



```
mov eax, 0x1234
jmp -0x5
```

LLVM

- Typical integration progression:
  - I have a project that compiles *something*
    - Need to make it faster or
    - Need a backend to actually produce native code.
  - Integrate with LLVM!

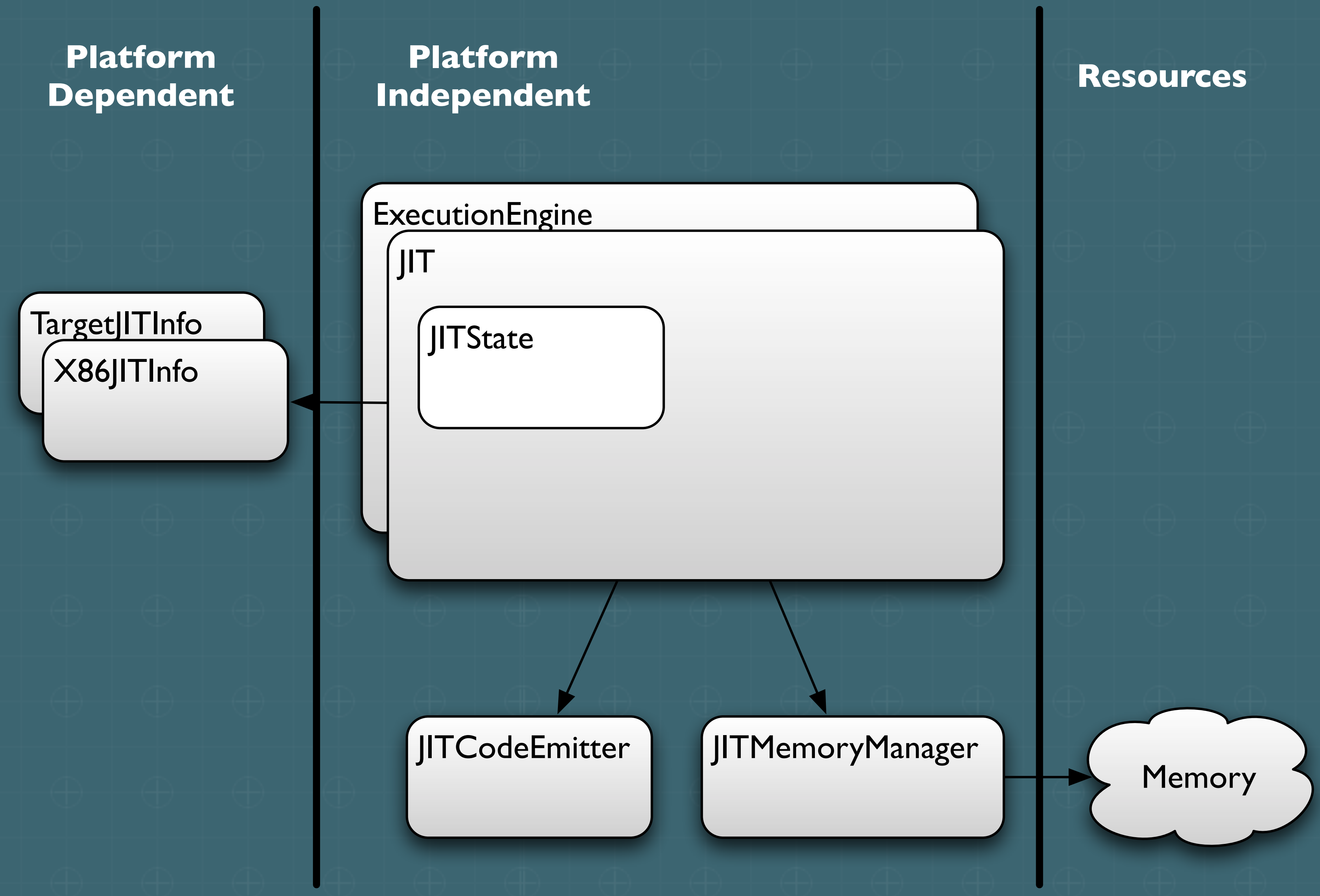
LLVM Integration

- “The LLVM JIT and You”
- Popular integration strategies
  - Emit IR directly, create a Module
    - MacRuby, GHC
  - Have your own VM instruction set, translate instruction by instruction to LLVM equivalents, then emit
    - Rubinius, ClamAV

LLVM JIT

- Assume a Module is created
- Connect a Module to an ExecutionEngine
- Request a handle to a function, ask the ExecutionEngine to run it
- ExecutionEngine emits code for the function, and stubs for all outgoing calls to non-emitted code

LLVM JIT


Platform Dependent
Platform Independent
Resources
ExecutionEngine
JIT
JITState
TargetJITInfo
X86JITInfo
JITCodeEmitter
JITMemoryManager
Memory

JITs and Security

- Compiling traditional executables is *typically* done by developers
- Code compilation is a trust boundary
  - You've accepted your vendor's code and binary
  - But now you're compiling *my untrusted code*

Incorrect Code Emission

- JITs don’t always produce perfect code
- Compiler bugs are often caught during development and testing
- What can happen when the JIT emits incorrect code?

Incorrect Code Emission

- Java x64 JIT *bug* patched on June 18th, 2011
- Intended code emission:

```
addq (%rsp),0xffffff2b  ; add 0xffffff2b to the value at %rsp
popfq                   ; pop 64 bits from stack, load
                        ; the lower 32 bits into RFLAGS
```

- Unintended code emission:

```
addq %rsp,0xffffff2b     ; shift the stack pointer!
popfq                    ; pop 64 bits from stack+0xffffff2b
                         ; load the lower 32 bits into RFLAGS
```

Incorrect Code Emission

- Many examples
  - Mozilla Bugzilla ID 635295 (Firefox 4.0 Beta)
    - Execution of an invalid branch due to an inline cache that existed for a free'd object
  - MS11-044 Microsoft .NET CLR JIT
    - The JIT produced code that confused an object as NULL or non-NULL
    - This was a great logic bug example!

Incorrect Code Emission

- What usually triggers them?
  - Use after free
  - Integer over/underflows (miscalculation of code paths)
  - Incorrect logic during code emission
- Are incorrect JIT code emissions a new bug class?
  - Depends on the root cause
  - Not for us to decide, but should be debated

JIT Primitives + Traditional Bugs

- JIT engines can be:
  - the source of vulnerabilities
  - a means to exploit them

matasano SECURITY

Exploitation Primitives

- JITs introduce unique exploitation primitives that would otherwise not be present in an application
  - JIT Spray
  - RWX Page Permissions
  - Reusable code sequences at predictable addresses

JIT Spray

- Dion Blazakis 2010
  - Flash ActionScript
- Create enough constants to contain native shell code, link together by semantic NOPs
- Transfer execution to mid-instruction, set up a stage 2, and begin executing
- I’m told by people smarter than me you can do it in 2 bytes with a short jmp

matasano SECURITY

JIT Spray

- JIT Spray in Firefox through JaegerMonkey
  - Not perfect, JaegerMonkey emits unoptimized code
  - Lots of bytes in the way we can’t control

```
var constants = [ 0x12424242, 0x23434343, 0x34444444, 0x45454545, 0x56464646,
0x67474747, 0x78484848, /test/ ]

0x40a05e: call   0x82d1820 NewInitArray ; create an array
0x40a063: mov    %eax,%edi              ; $edi holds returned array object
0x40a065: mov    0x24(%edi),%edi        ; load obj->slots in to $edi
0x40a068: movl   $0xffff0001,0x4(%edi)  ; JSVAL_TYPE_INT32 to object->slots[1]
0x40a06f: movl   $0x12424242,(%edi)     ; 1st constant into object->slots[0]
0x40a075: mov    %eax,%edi
0x40a077: mov    0x24(%edi),%edi
0x40a07a: movl   $0xffff0001,0xc(%edi)
0x40a081: movl   $0x23434343,0x8(%edi)  ; 2nd constant
0x40a088: mov    %eax,%edi
0x40a08a: mov    0x24(%edi),%edi
0x40a08d: movl   $0xffff0001,0x14(%edi)
0x40a094: movl   $0x34444444,0x10(%edi) ; 3rd constant
0x40a09b: mov    %eax,%edi
0x40a09d: mov    0x24(%edi),%edi
0x40a0a0: movl   $0xffff0001,0x1c(%edi)
0x40a0a7: movl   $0x45454545,0x18(%edi) ; 4th constant
```

JIT Spray

- JIT Spray in Firefox through TraceMonkey
  - Floating point games
  - -6.828527034422786e-229 = 0x9090909090909090
    - 0x90 = x86 NOP instruction

```
var a = -6.828527034422786e-229;
var b = -6.828527034422786e-229;
var c = -6.828527034422786e-229;
var d = -6.828527034422786e-229;

0x429eda: movl   $0x90909090,0x5c0(%esi)
0x429ee4: movl   $0x90909090,0x5c4(%esi)
0x429eee: movl   $0x90909090,0x5c8(%esi)
0x429ef8: movl   $0x90909090,0x5cc(%esi)
0x429f02: movl   $0x90909090,0x5d0(%esi)
0x429f0c: movl   $0x90909090,0x5d4(%esi)
0x429f16: movl   $0x90909090,0x5d8(%esi)
0x429f20: movl   $0x90909090,0x5dc(%esi)
```

Memory Protections

- Nearly all JITs we surveyed produce RWX pages
  - Weakens DEP
  - Breaks assumptions behind copy-on-write mirror pages
    - Knowledge of both RW/RX pages not required
  - Blind Execution
    - Overwrite RWX JIT page contents
    - Trigger the original JIT'd script
  - This isn't going away for Inline Cache designs without some performance impact

matasano SECURITY

Memory Protections

- RWX pages can be reused
  - Array index read/write
    - Point into JIT page
      - Write raw shell code, trigger JavaScript
      - Read branch addresses back to C++ in a DLL
  - Overflows
    - Heap overflow in adjacent RW page

Firefox 5.0

```
02808000-0280c000 rw-p  Read/Write Heap memory
0280c000-0281c000 rwxp  Read/Write/Execute JIT page
```

  - ROP
    - No need to find that VirtualAlloc stub

gaJITs

- ROP Gadgets are small sequences of code found in an existing DLL or .text
    - Combine them to get arbitrary code execution
- Predictable instructions on JIT pages at static offsets
- JIT's produce lots of native code
    - You aren't constrained to just one library mapping
    - Does not require controllable constants like JIT Spray

gaJITs

- Finding usable gaJITs depends on the JIT design
  - *ret* or branch-based control flow?
  - inline caching
  - (in)frequent calls to C++ stubs
- How does script function A get turned into native code B where native code B contains gaJIT X
  - Requires the right source code to generate them
  - Requires a specific gaJIT-finding tool

JIT Feng Shui

- Our version of Heap Feng Shui... except for JITs
  - Heap Feng Shui
    - Alex Sotirov 2007
    - Influence the heap layout via JavaScript
  - JIT Feng Shui
    - Untrusted input influences JIT output
    - Specific inputs create predictable code patterns
  - We could have called it jiuJITsu..

matasano SECURITY

JIT Feng Shui

- Controlling register contents with a TraceMonkey gaJIT

  ```
  gaJIT at offset 0x9e18 (10 matches)
  pop esi ; pop edi ; pop ebx ; pop ebp ; ret
  ```

- LLVM
  - Portable shellcode!

JIT Feng Shui + gaJITs

- Circumvents constant masking
  - Defeated by NOP padding
  - Much harder with allocation restrictions
- Difficult and noisy
  - Requires a JIT spray to map enough pages
- Not researched on other JITs / architectures yet

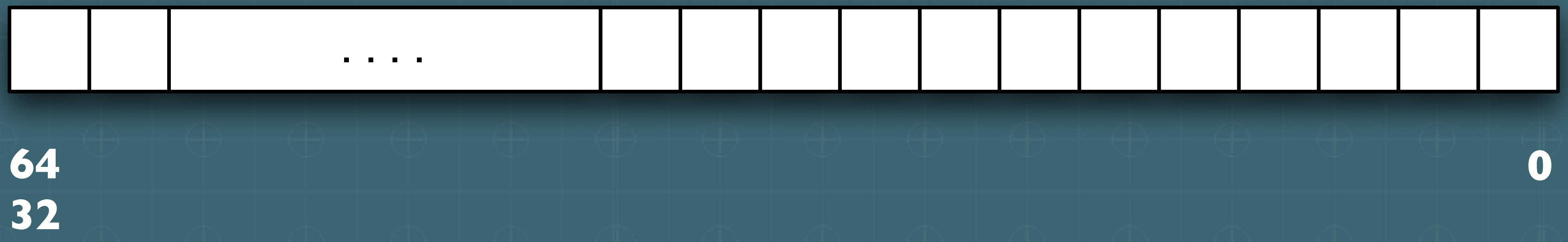
JIT Protections

- The OS provides some basic protections to the process
  - (ASLR) Address Space Layout Randomization
  - (DEP) Data Execution Prevention
  - Code Signing
  - JITs can negate these by design
- JIT engines have no control over their input
  - ... but completely control their output

Emission Randomization

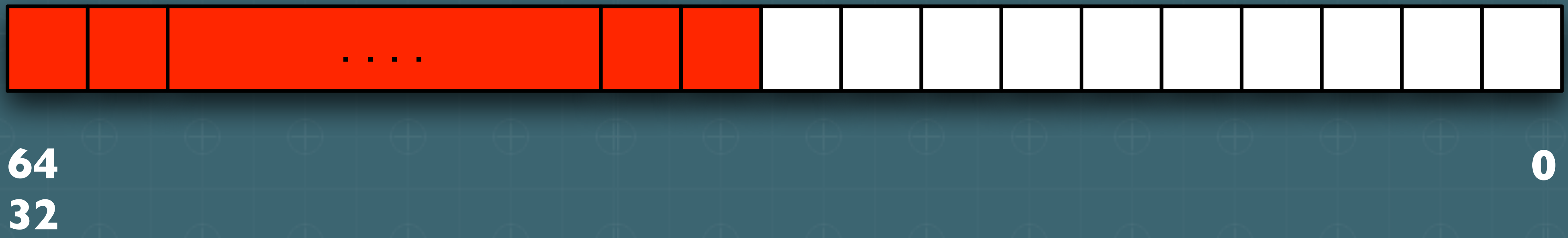
- Memory for emission is allocated via mmap or VirtualAlloc
  - VirtualAlloc is *not* randomized by default
    - You can request the address you want mapped
    - V8 and IE9 do this
  - mmap on Linux randomizes anonymous mappings
- Extend ASLR to compiler-allocated memory

Randomization



Randomization

Allocation Randomization

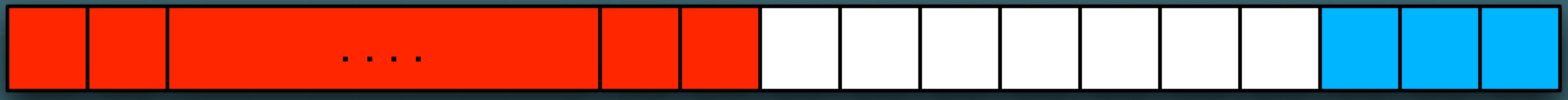


Randomization

- Intra-page offsets (bottom 10 bits) are still predictable
- Since you're emitting code, you can shift each function emitted by inserting NOPs

Randomization

Allocation Randomization **NOP Padding**



64
32

0

Randomization

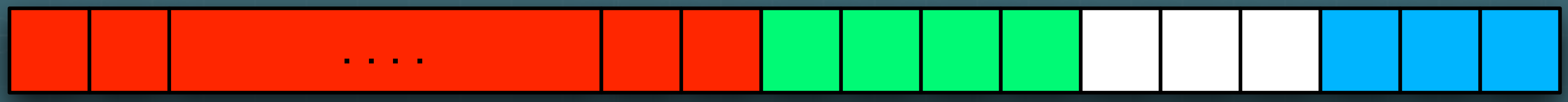
- Function emission is still predictable
- If you're batching the functions you're emitting, you can shuffle the order at which they're produced

Randomization

Allocation Randomization

Function Shuffling

NOP Padding



64
32

0

Guard Pages

- Firefox 5.0 adjacent heap and JIT pages

```
02808000-0280c000 rw-p  Read/Write heap memory
0280c000-0281c000 rwxp  Read/Write/Execute JIT page
```

- If an overflow occurs in the first RW heap mapping, an attacker can write native code into the RWX page
- Guard pages prevent heap overflows from writing to RWX JIT pages

```
02808000-0280c000 rw-p  Read/Write heap memory
0280c000-0281c000 r--p  Read Only memory
0281c000-0282c000 rwxp  Read/Write/Execute JIT page
```

Constant Folding

- 4-byte constants allow room to insert instructions on x86
- Chained 4-byte chunks allows for a stage 1 payload
- Solution: Fold large constants into 2-byte maximum constants and reassemble at runtime.
- *Problem:* If the instructions are predictable an attacker can bypass this by injecting the right constants
- V8 did this for a while, now they use constant blinding

Constant Blinding

- XOR all untrusted immediate values by a secret cookie
- Generate a random value at startup
  - untrusted immediate ⊕ secret cookie
- Emit code that XORs the value at runtime

```
                               mov eax, 0x84521310
xor eax, 0x00112233   →
                               xor eax, 0x84433123
```

Allocation Restrictions

- JIT Spray requires mapping a lot of memory
- Capping the number of pages helps mitigate this attack
- For language runtimes, some info about code can be known ahead of time
  - code size
  - libraries used
- Unfortunately, this protection mechanism makes more sense for browsers than language runtimes

matasano SECURITY

JIT Comparison

	V8	IE9	Jaeger Monkey	Trace Monkey	LLVM	JVM	Flash / Tamarin
Secure Page Permissions	✗	✓	✗	✗	✗	✗	✗
Guard Pages	✓	✗	✗	✗	✗	✗	✗
JIT Page Randomization	✓	✓	✗	✗	✗	✗	✗
Constant Folding	✗	✗	✗	✗	✗	✗	✗
Constant Blinding	✓	✓	✗	✗	✗	✗	✗
Allocation Restrictions	✓	✓	✗	✗	✗	✗	✗
Random NOP Insertion	✓	✓	✗	✗	✗	✗	✗
Random Code Base Offset	✓	✓	✗	✗	✗	✗	✗

JIT Comparison

- IE9 doesn't require guard pages
- Tamarin/TraceMonkey (NanoJIT) implemented random NOP padding but forgot to enable it
- Guard pages in Chrome are brand new as of 8/4/2011
- As a result of our research, Firefox should be implementing some of these very soon

jitter

- jitter is our toolchain for:
  - Tracing JIT code emission
  - Tracking JIT memory permissions
  - JIT Fuzzer coverage
  - Searching for gaJITs
- Implemented as a set of Nerve scripts
  - Uses ragweed debugging framework
  - We also wrote a native Java JIT hook

jitter

- Support for LLVM and Firefox JITs
  - Nerve breakpoint files for specific JIT hook points
  - Interact with the process at each breakpoint with Ruby
  - Extract arguments, data, instructions
- Generic script for tracking JIT page allocations
  - Just needs a list of call sites
  - Can be used to start support of new JIT engines
- gaJIT finder is built-in
  - Receives an array of JIT pages
  - Output locations for repeated gaJITs
  - Easily repurposed for other ROP tools

matasano SECURITY

fuzzer(s)

- Fuzzing JIT engines is difficult
  - Testcases must have valid syntax
  - Multiple components before you hit the JIT
- Rubinius Fuzzer (LLVM JIT)
- JavaScript grammar fuzzer (Firefox JITs)
- Fuzzer driver framework

fuzzing bitcode

- We attempted to fuzz LLVM bitcode directly
- Dumb-fuzzing at first
  - Way too many coredumps to go through
- LLVM's BitcodeReader was *not* designed with security in mind
- Found a parsing bug; submitted patch

rubyfuzz

- Ruby fuzzer for targeting Rubinius
  - Generated Ruby code from a subset of Ruby grammar
  - Avoided Rubinius VM to target other Ruby implementations
    - MacRuby, JRuby, YARV, MRI, etc
- Fuzzer driver also in Ruby (Hoke)

rubyfuzz

- Modeled Ruby grammar as Ruby objects
  - Terminals → Arrays
  - Non-terminals → Generators
- Permuted method invocations, block definitions, block invocations and other Ruby constructs
- Seeded with common Ruby idioms

JavaScript Fuzzer

- JavaScript Grammar fuzzer for Firefox JITs
- Targets the JIT and interpreter only; not the DOM
- Describe JavaScript in flat text files
    - types, methods, properties, keywords, and operators
- Parse text files and serialize into Ruby OpenStruct
- Iterate over the grammar
    - Follow JSOP bytecode instructions to
        - Fast Paths
        - Inline Caches
        - C++ Stubs
- Hundreds of millions of iterations through ./js

A bug our fuzzer found

- Our fuzzer found a critical bug in SpiderMonkey

```
a = new Array();
a.length = 4294967240;
b = function bf(prev, current, index, array) {
    document.write(current);
    current[0] = "hello";
}
a.reduceRight(b, 1, 2, 3);
```

- Info Leak: read arbitrary data from *current*
- Code Execution: call a method on *current*

fuzzer(s)

- A note on fuzzing for info leaks
  - Fuzzing should be fast
  - Instrumentation to monitor individual memory access is slow
- Differential fuzzing for info leaks
  - Can be generalized to multiple implementations of any language spec
  - Two JavaScript implementations
    - d8 (v8) / js (Mozilla)
    - Feed them identical testcases
    - Record the output
      - What is the expected output type/value?

matasano SECURITY

Questions

?

matasano SECURITY