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

matasano SECURITY

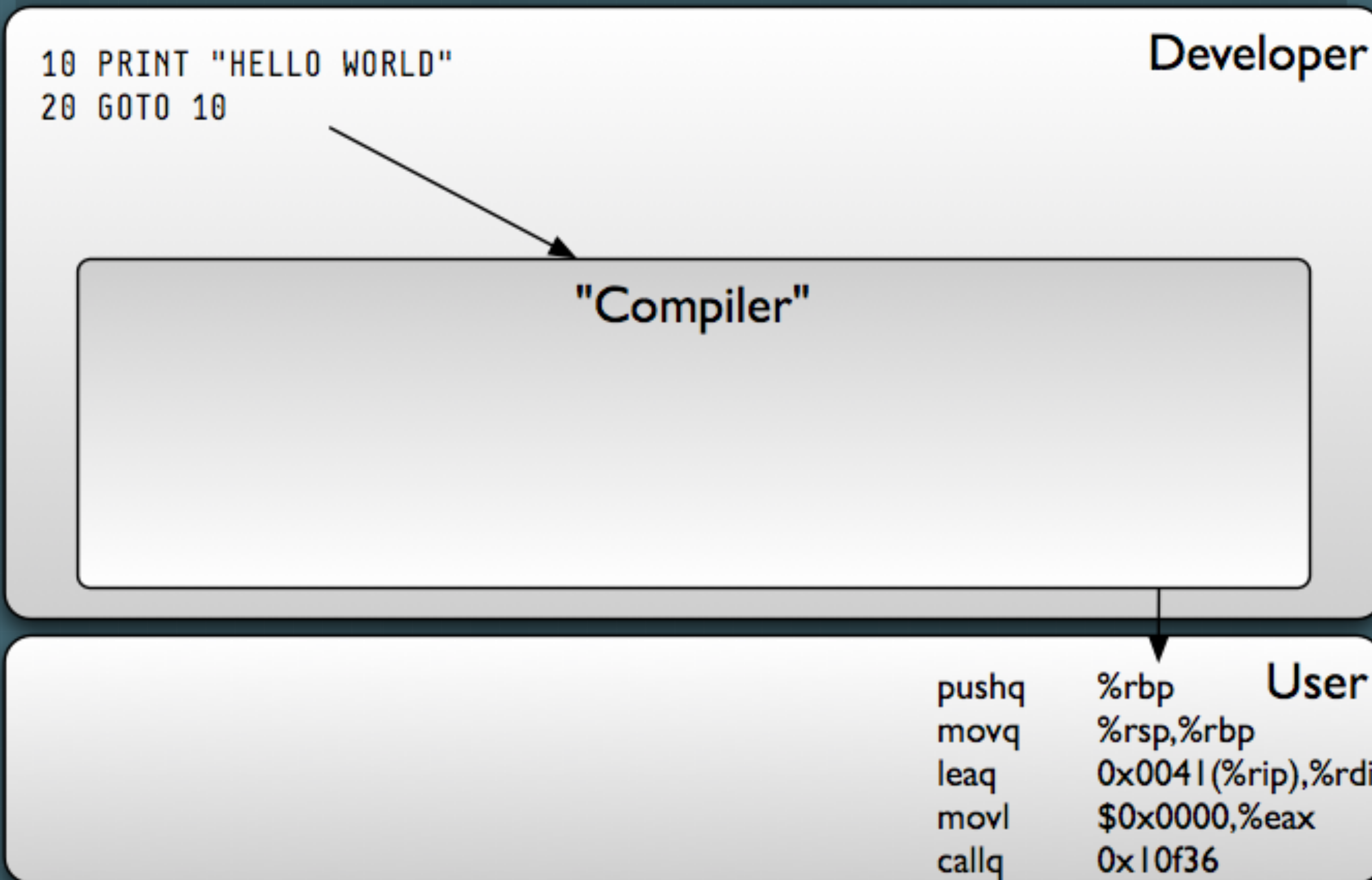
Overview

- 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

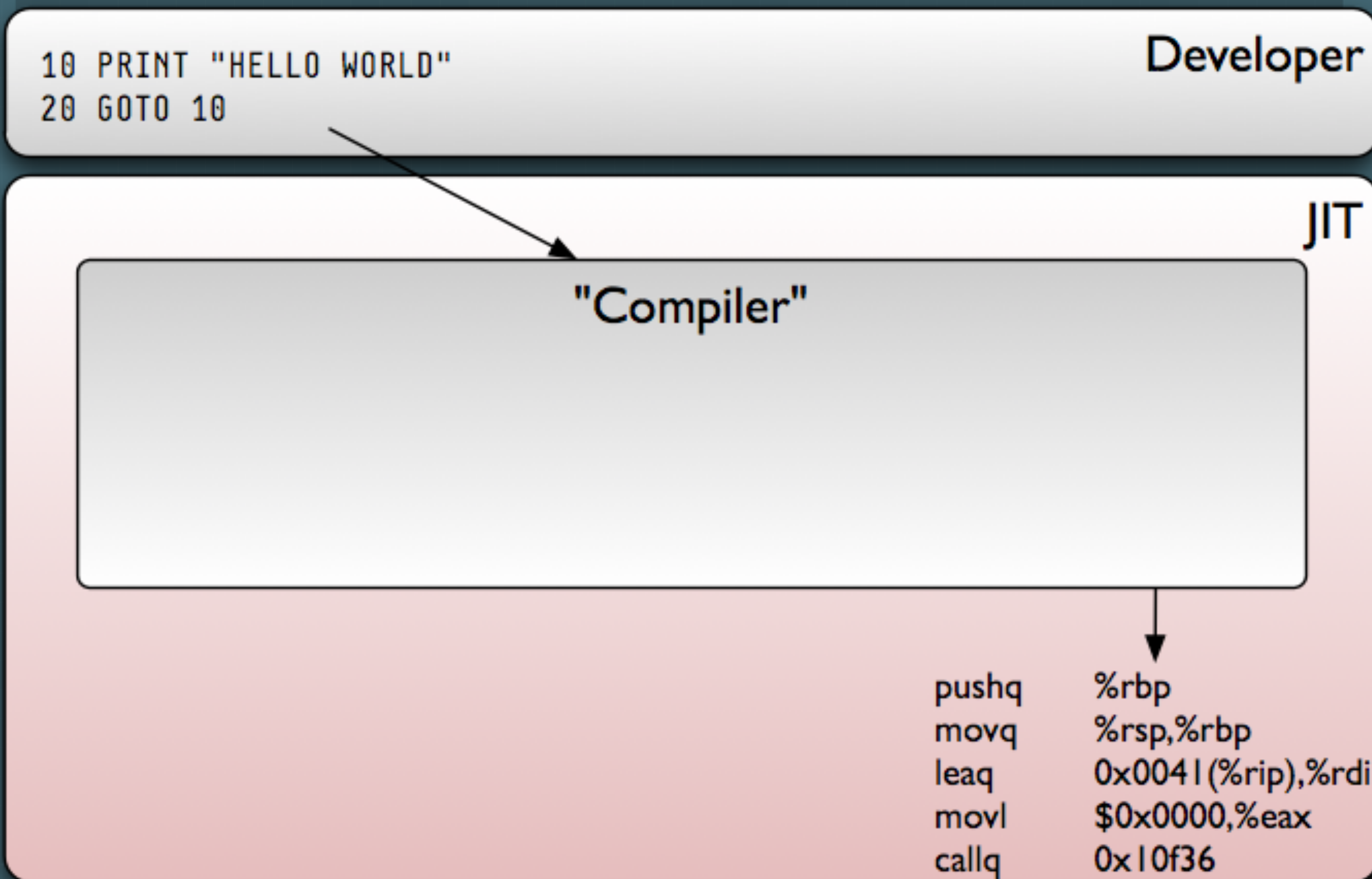
- Interpreters and JIT Engines
  - Parse high level languages
  - Generate bytecode
  - Optimize and compile bytecode to native code
- They are everywhere
  - Browsers
  - Language runtimes (Java, Ruby, C#)

Introduction to JITs

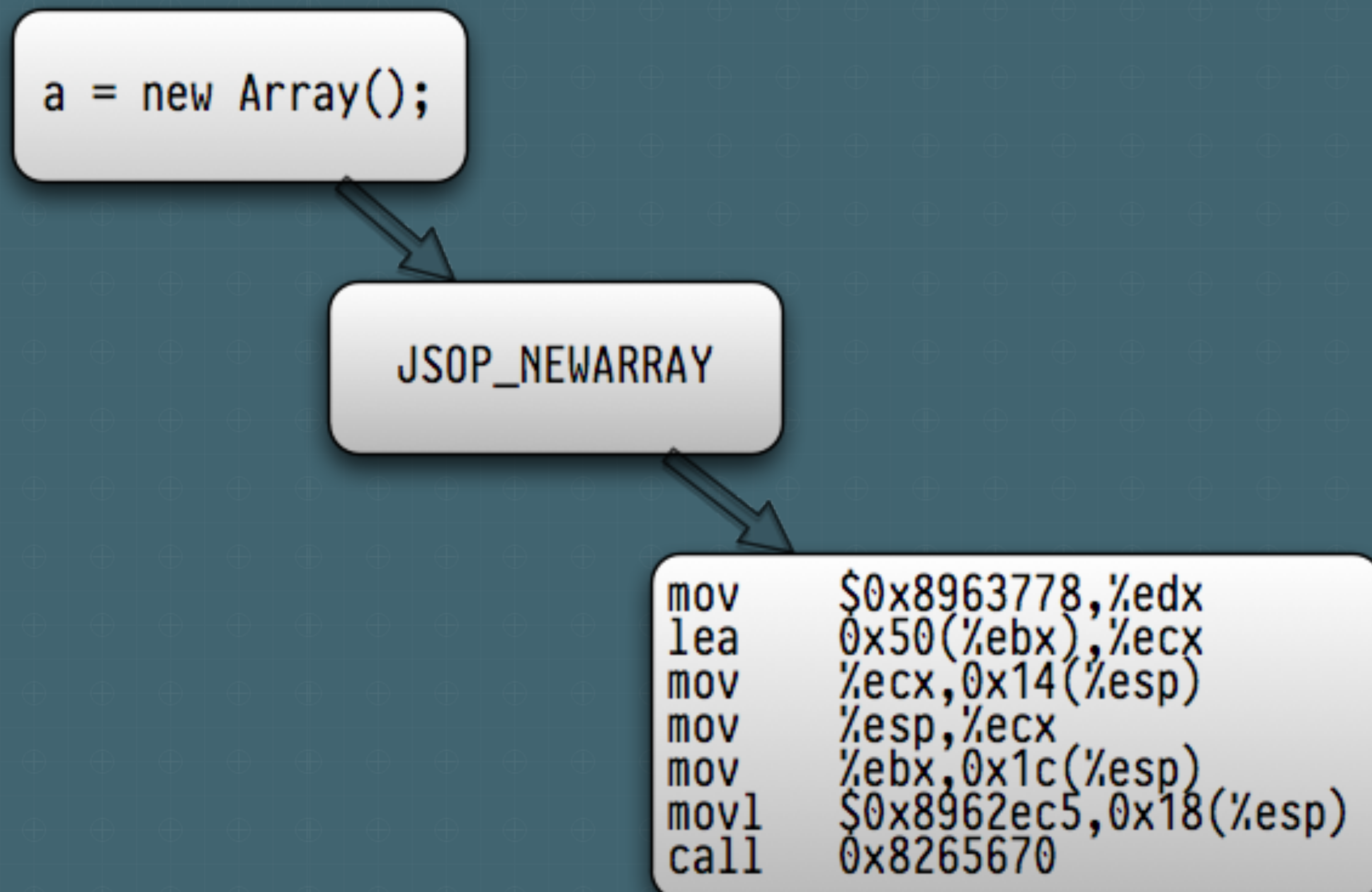


Introduction to JITs

Introduction to JITs



Introduction to JITs



matasano SECURITY

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
        - Trigger interpreter

Introduction to JITs

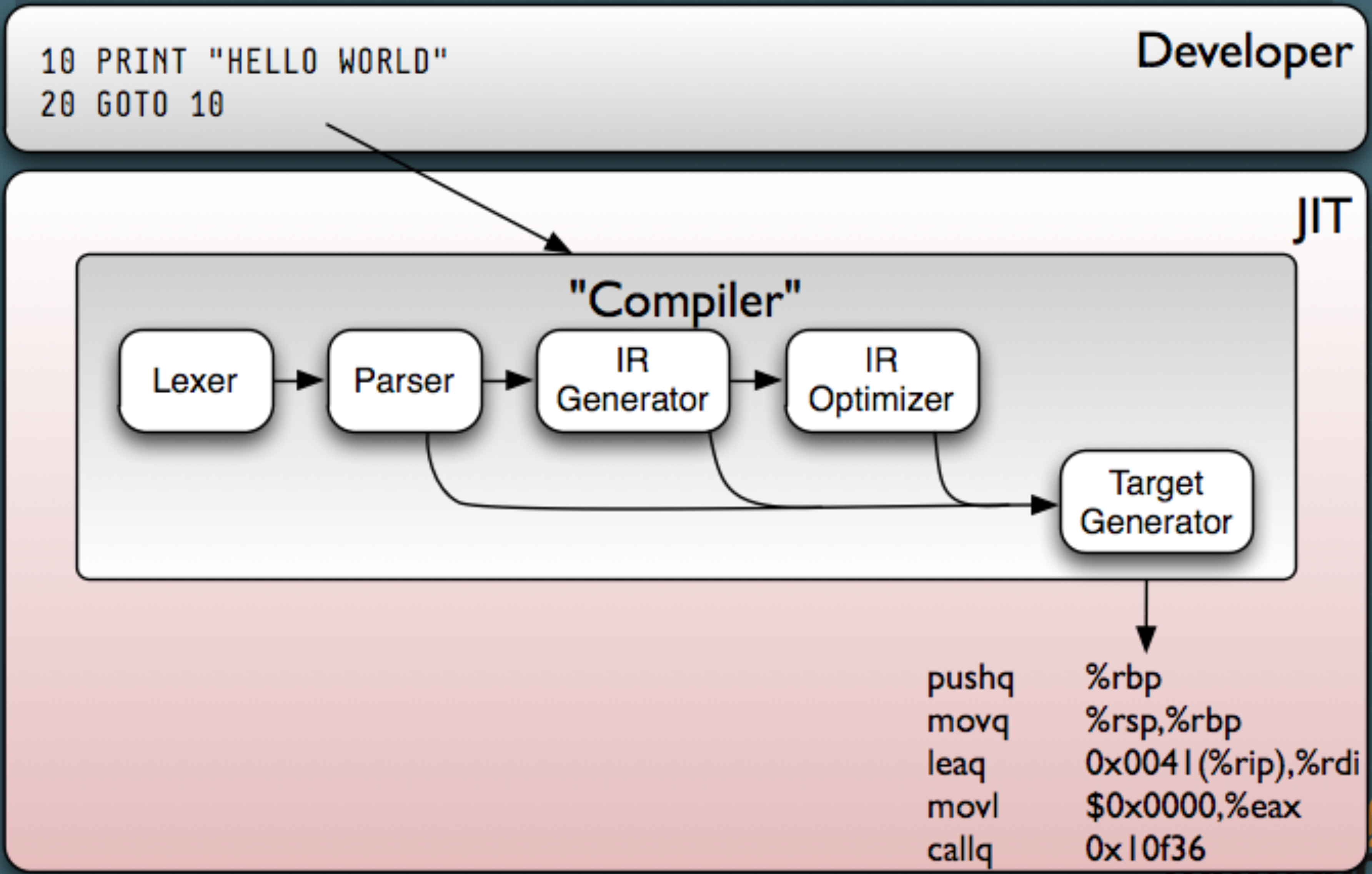
- Untrusted bytecode
  - Can be delivered from untrusted sources
    - Flash, CLR, LLVM
  - Completely external to the compiler

matasano SECURITY

Introduction to JITs

- Trusted bytecode
  - Produced internally by a trusted front end
    - Spidermonkey
  - Still potentially usable to an attacker with control of the process

Introduction to JITs



Introduction to JITs

- JITs come in a few different designs
  - Method
    - Firefox
    - LLVM
    - V8
  - Tracing
    - Firefox
    - Rubinius

matasano SECURITY

Introduction to JITs

- Tracing
  - Only JITs CPU-intensive code
  - Enables optimizations
  - Types are generally known from tracing

matasano
SECURITY

TraceMonkey

- Introduced in Firefox 3.5
- Tracing JIT
- Uses NanoJIT as an assembler
- SpiderMonkey Bytecode LIR Bytecode
- NanoJIT LIR Native code

matasano SECURITY

TraceMonkey

- TraceMonkey JITs hot code blocks
  - The recorder traces execution of SpiderMonkey IR
  - Produces trace trees
  - Emits optimized code
- Doesn't handle type changes well

TraceMonkey

- CodeAlloc Class
  - Handles allocating JIT pages that will hold code
- CodeList Class
  - Inline meta-data for tracking the location of JIT pages

Introduction to JITs

- Method
  - JITs entire functions/blocks
  - Usually generates unoptimized code
  - 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 new types

JaegerMonkey

- Inline Caching
  - JavaScript is dynamically typed
  - It can JIT a generic function that handles multiple types

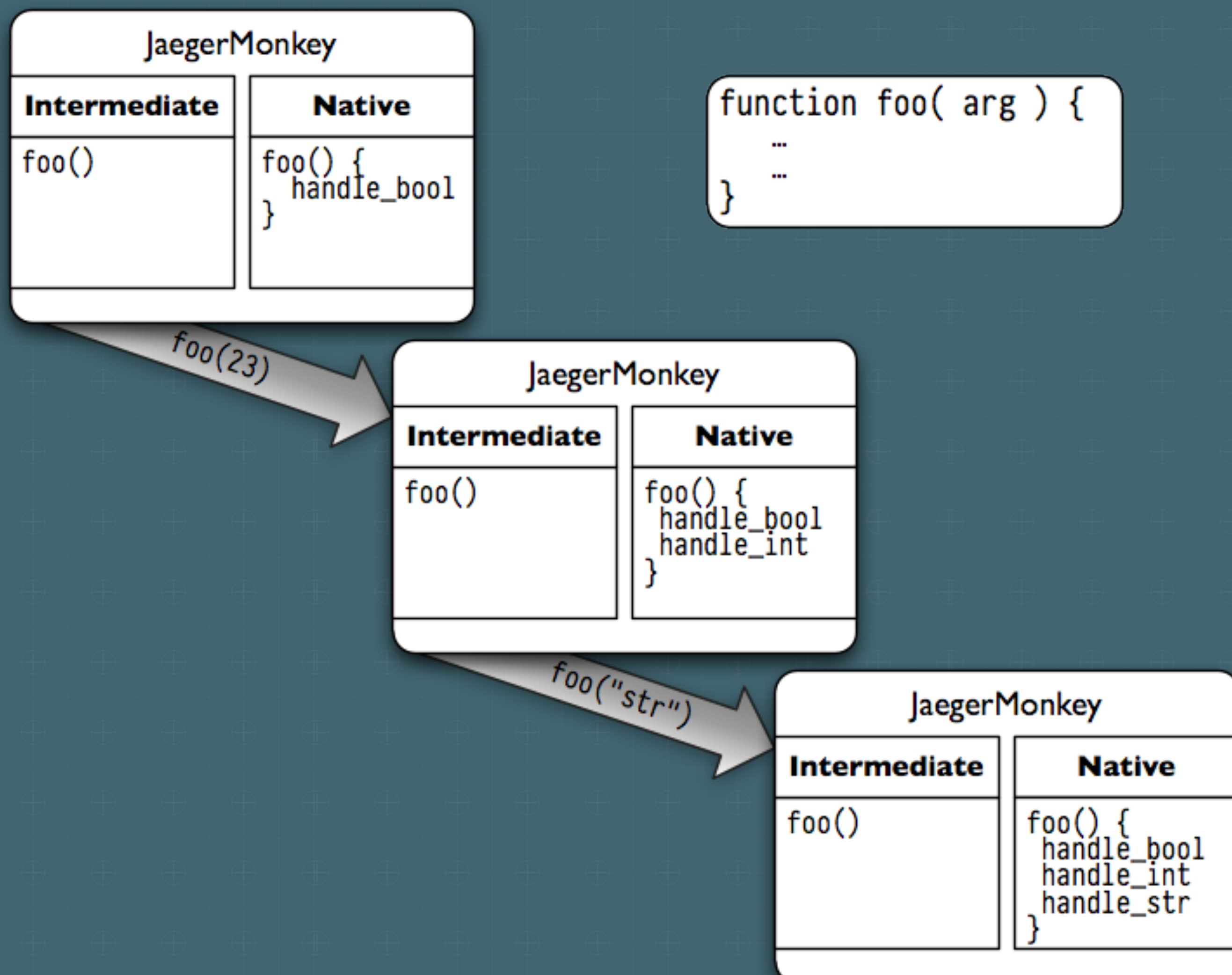
```
function a = blah(var b){
   for(i=0;i<10;i++) {
     b += i;
   }
}
```

  - The JIT emitted code needs to be rewritten for each new type it encounters

```
blah("hello")
blah([0,1,2,3])
```

    - The Inline Cache makes this possible

JaegerMonkey



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 calls into C++ code from JIT pages
  - Typically exist to augment a fast path

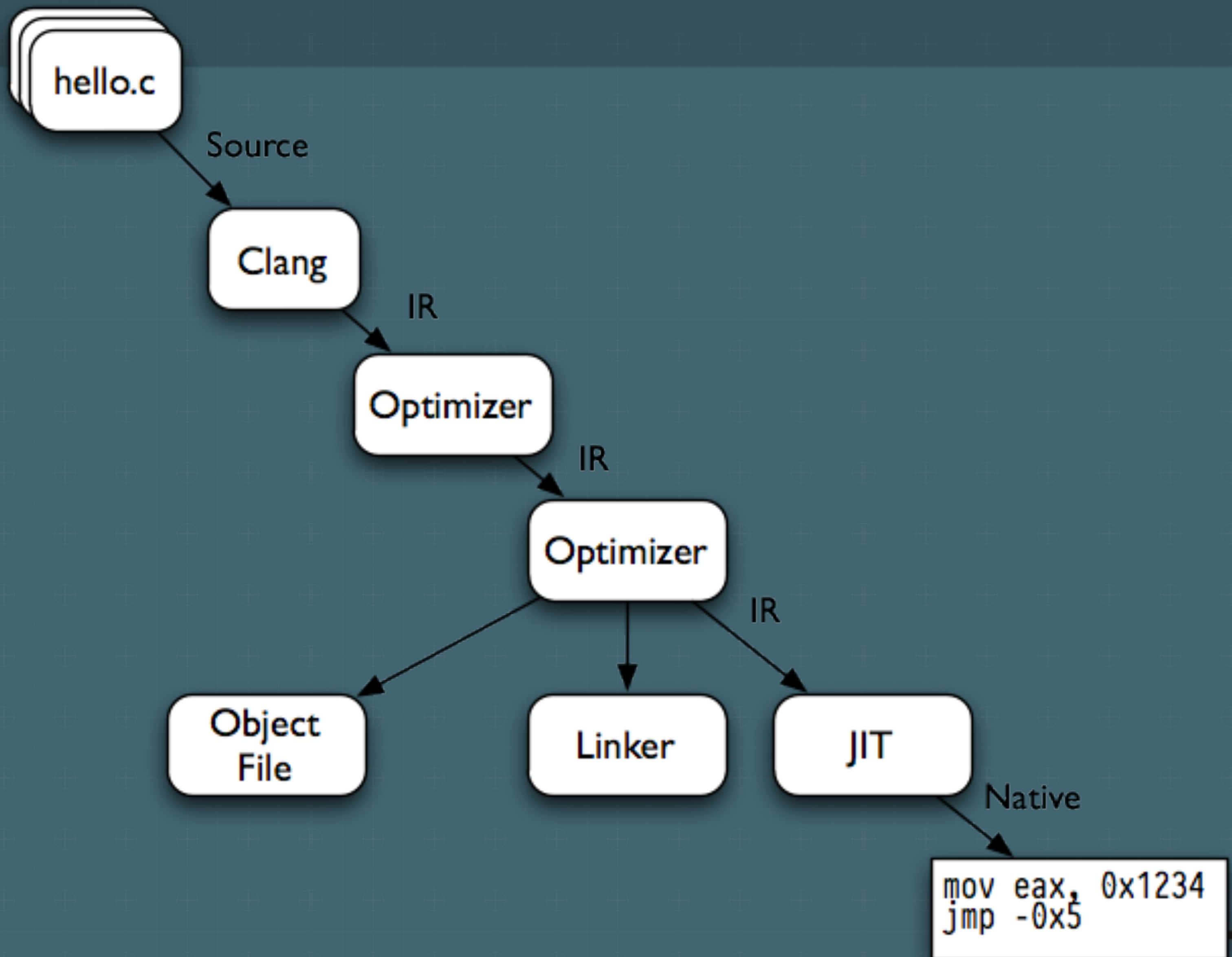
JaegerMonkey

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

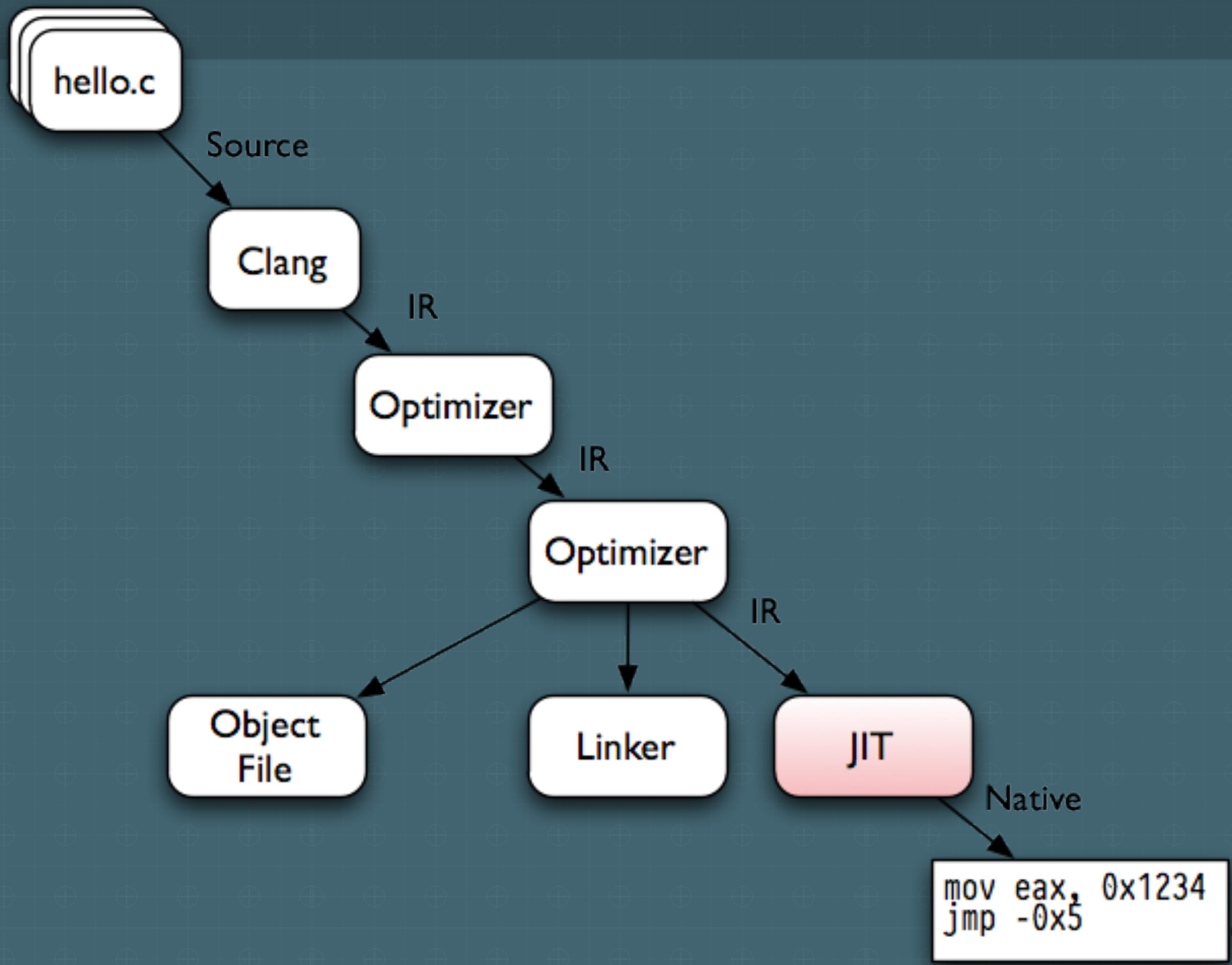
LLVM

- Originated in UIUC
- A suite of libraries from the start
- Initially used GCC as a front end
- Now supports C, C++ and Objective-C entirely
- Frontends available for
  - Python, Ruby, Haskell, PHP, etc

LLVM


hello.c
Source
Clang
IR
Optimizer
IR
Optimizer
IR
Object File
Linker
JIT
Native
mov eax, 0x1234
jmp -0x5

LLVM



hello.c
Source
Clang
IR
Optimizer
IR
Optimizer
IR
Object File
Linker
JIT
Native
mov eax, 0x1234
jmp -0x5

LLVM

- Typical progression:
  - I have a project that compiles *something*
  - Need to make it faster
  - Integrate with LLVM!

LLVM Integration

"The LLVM JIT and You"

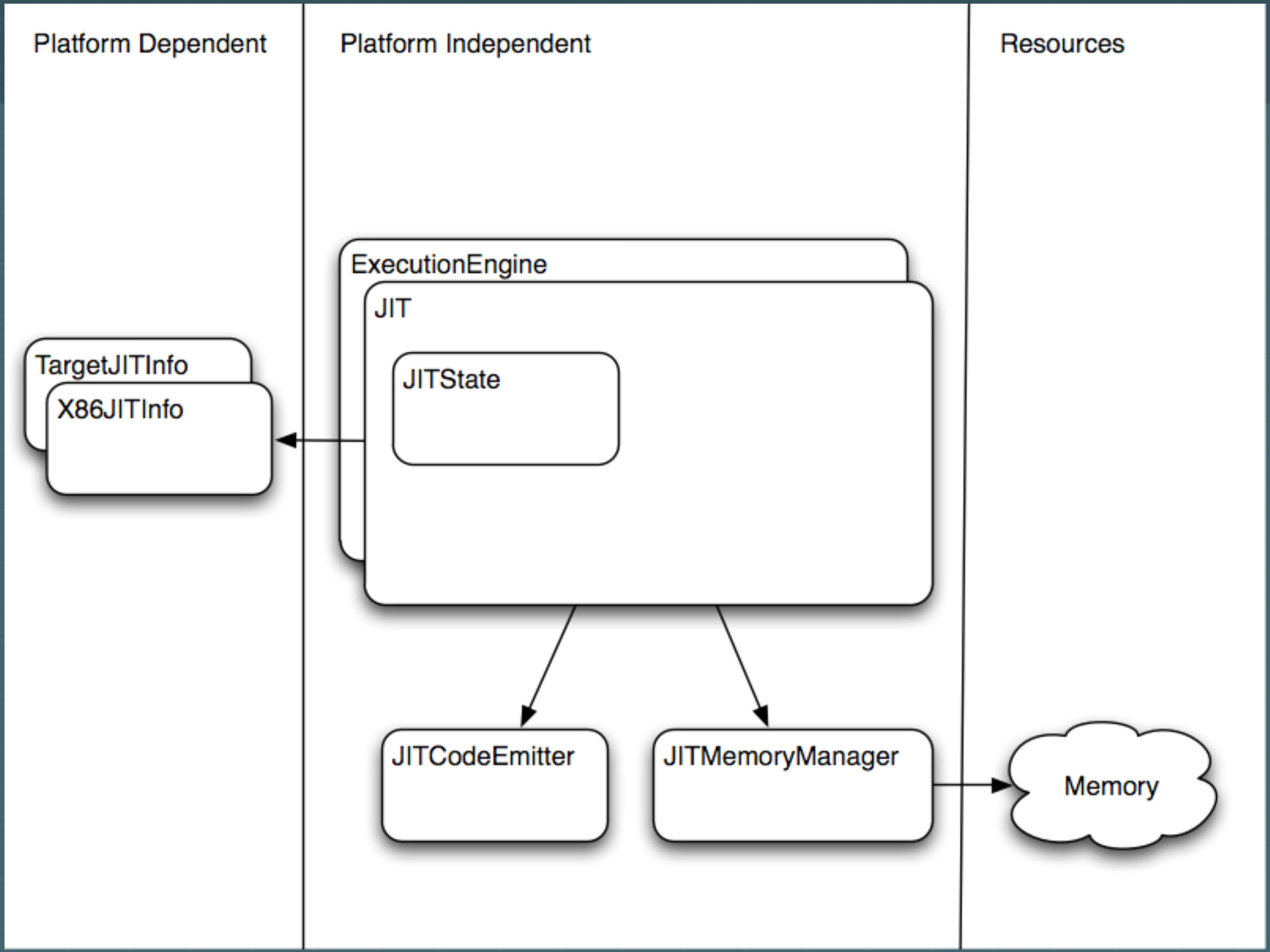
- Popular integration strategies
- Emit IR directly, create a Module
    - MacRuby, GHC
- Have your own VM instruction set, translate instruction by instruction to LLVM equivalents, then emit
        - Rubinius, ClamAV

LLVM JIT

- Assume a Module is created
- Connect a Module to an ExecutionEngine
- Request a handle to a function, ask the ExecutionEngine for it
- ExecutionEngine emits code for the function, for all data it references and emits stubs for all functions it calls

matasano SECURITY

LLVM JIT



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

LLVM Attack Scenarios

- Bitcode injection.
  - Portable shellcode!
- Bitcode parser flaws
  - Shouldn't be trusted, but will probably be
- Poor emission
  - Incorrect code being emitted
- Liberal Memory Protections
  - Common vector in general
- Mistranslated instruction set

JITs and Security

- Compiling traditional executables is *typ* y developers
- Code compilation is a trust boundary
  - You've accepted your vendor's code and binary
  - But now your compiling *my untrusted code*

Incorrect Code Emission

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

matasano SECURITY

Incorrect Code Emission

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

addq %rsp,0xffffff2b ; shift the stack pointer!

popfq ; pop 64 bits from stack+0xffffff2b ; load the lower 32 bits into RFLAGS

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

matasano SECURITY

Incorrect Code Emission

- There are many examples of this
  - 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!

matasano SECURITY

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 it should be debated by the security research community

matasano SECURITY

JIT Primitives + Traditional Bugs

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

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

- Publicized by Dion's talk
- 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
- Only 2 JIT engines randomize *VirtualAlloc*
  - V8, IE9
  - Does it matter on 32 bit?

JIT Spray

- JIT Spray in Firefox through JaegerMonkey
  - 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 into 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**

Memory Protections

- Nearly all JITs we surveyed produce RWX pages
  - Breaks DEP / W^X
  - Breaks assumption behind 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

matasano SECURITY

Memory Protections

- RWX pages can be repurposed for different things
  - Array index read/write
    - Point into JIT page
      - Write raw shell code, trigger JavaScript
      - Read pointers to a DLL that may be mapped beyond your reach
  - Overflows
    - Heap overflow in adjacent RW 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
- gaJITs: Predictable instructions on JIT pages at predictable 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

matasano SECURITY

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 inputs to generate them
  - Requires a specific gadget-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 creates predictable code patterns
- We could have called it jiuJITsu...

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!

matasano SECURITY

JIT Feng Shui + gaJITs

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

matasano SECURITY

JIT Protections

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

Emission Randomization

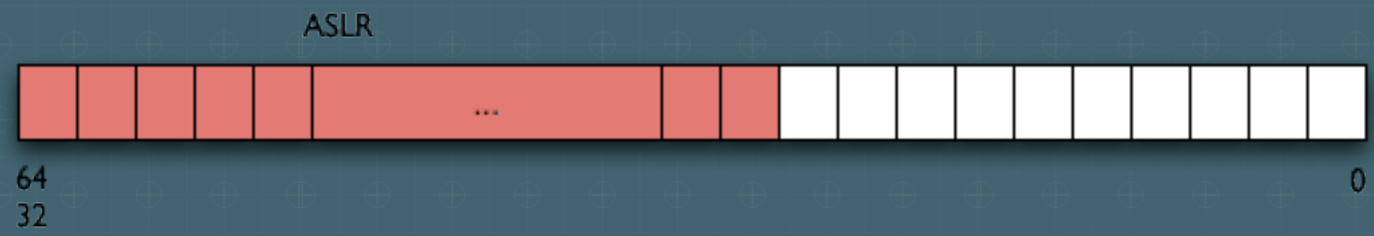
- Emitting code at runtime allows for randomizing addresses
  - High 22 bits (54 on 64 bit [fact check]) get randomized due to ASLR, or opting into a random address
  - Low ~3 bits get randomized due to NOP insertion
  - The middle bits get optimized within code region

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

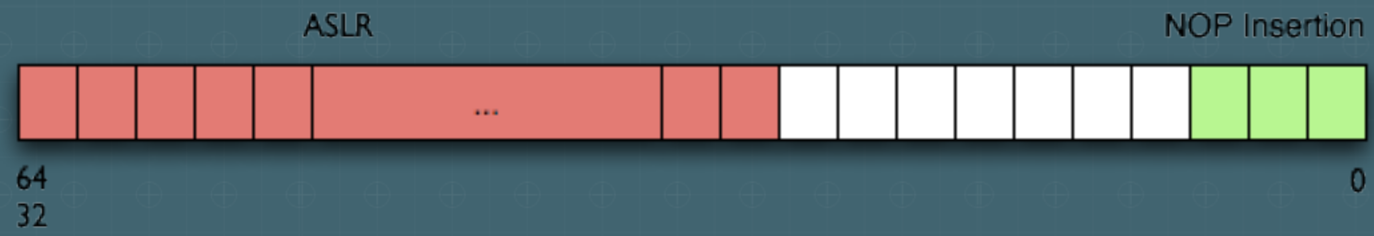
Randomization



Randomization

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

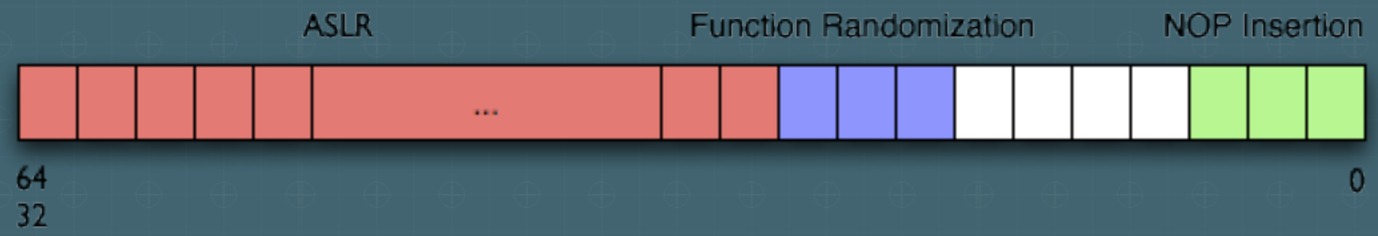
Randomization



Randomization

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

Randomization



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
  - Trigger JavaScript to get code execution
    - Blind execution
  - No need to gain EIP or overwrite a function pointer

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 allows 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 awhile, now they use constant blinding

matasano SECURITY

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

```
xor eax, 0x00112233
```

→

```
mov eax, 0x84521310
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

JIT Comparison

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

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 the 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, code, 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
  - Outputs locations of repeated gaJITs
  - Easily repurposed for other ROP tools

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)
  - Targets the JIT and interpreter only
    - DOM bugs are boring

matasano SECURITY

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 framework Hoke
  - Generated test cases external to the Ruby implementation

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 constructs

matasano SECURITY

fuzzer(s)

- JavaScript Grammar fuzzer for Firefox JITs
- Describe JavaScript in flat text files
  - types, methods, properties, keywords, operators
- Parse text files and serialize into Ruby OpenStruct
- Iterate over the grammar
  - Fast Paths
  - Inline Caches
  - C++ Stubs
- Hundreds of millions of iterations through ./js

fuzzer(s)

- A note on fuzzing for info leaks
    - Fuzzing should be fast
    - Instrumentation is slow
- Differential fuzzing for info leaks
    - Two JavaScript implementations
        - d8 (V8) / js (Mozilla)
        - Feed them the same testcase
        - Record the output
            - What is the expected output type/value?
- Can be generalized to multiple implementations of any language

matasano SECURITY

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);
}
a.reduceRight(b, 1, 2, 3);
```

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

Demo?

Questions

?

matasano
SECURITY

Please complete the Speaker Feedback Surveys. This will help speakers to improve and for Black Hat to make better decisions regarding content and presenters for future events.

blackhat usa+2011

EMBEDDING SECURITY

matasano SECURITY