Editors: Dino A. Dai Zovi, ddz@theta44.org | Alexander Sotirov, alex@sotirov.net

The Security Challenges of Client-Side Just-in-Time Engines

Chris Rohlf | Leaf SR
Yan Ivnitskiy | Matasano Security

Shortly after the Web's birth, simple, static, and mostly textual documents were the norm, and a simple browser sufficed for practically all use. As the Web grew to become a nascent commerce platform, developers began seeing the need to push logic to browsers to perform validation, increase interactivity, or make application functionality appear and disappear.

After some early competition between Netscape and Microsoft, JavaScript ended up becoming the language of choice for writing client-side code. JavaScript borrows its syntax from languages such as C and its object model from Self, giving it a loose dynamic typing system. Early implementations provided straightforward JavaScript interpreters that tried to do little more than parse and execute each statement. That, however, changed over the last decade as two trends took hold: the exponential increase in Web application complexity and prevalence, and the commercial advantage of controlling browser market share. Arguably, the prime differentiator for users became Web application performance, and JavaScript execution speed became the sole metric in the browser wars. Enter just-in-time (JIT) engines.

JIT engines generate native code from a higher-level representation at runtime to obtain better performance while maintaining portability. As with most ideas in compiler technology, the basic concept has existed for some time, tracing its history back to the '60s. Popularized by Smalltalk, JIT engines are now prevalent in major-language runtimes such as the Java virtual machine (JVM) and .NET virtual machine. JavaScript JIT engines began their foray into browsers with Firefox's TraceMonkey and now exist in every major browser and some plug-ins, with Google Chrome's V8, Internet Explorer's Chakra, Opera's Carakan, and Safari's JavaScriptCore. Browser JIT engines have become so important that Mozilla has developed and shipped three separate ones, each with its own approach: TraceMonkey (recently removed from the latest versions of Firefox), JaegerMonkey, and IonMonkey.

So, why are browser JavaScript engines interesting from a security perspective? First, any added complexity in a software system will increase the possible program states, introducing a larger attack surface and the possibility of more exploitable flaws. JIT engines, however, alter the environment in which they execute in far more interesting ways, not only through implementation flaws but also by their fundamental operation modes.

JIT Engine Weaknesses

Modern OSs take many precautions to prevent maliciously introduced code from executing as a result of an exploited vulnerability. These vulnerabilities manifest themselves when a program fails to properly check data accesses and inadvertently permits exploit code to overwrite data that manages control flow, transferring execution to malicious code. When DEP (Data Execution Protection) is enforced, a process's memory ideally is partitioned into two types: readable and writable for data, and readable and executable for code. Once code is loaded into memory, it shouldn't be overwritten by malicious instructions. Likewise, pages of memory that are controllable by untrusted parties shouldn't contain executable code. Because a large class of exploits relies on predictable memory addresses, ASLR (address space layout randomization) shuffles a

Copublished by the IEEE Computer and Reliability Societies

process's address space at runtime, reducing the odds of a successful attack. Time has shown the combination of DEP and ASLR to be an effective defense against memory safety vulnerabilities.

As part of a JIT engine's normal operation, memory must be allocated, written to, and executed routinely, side-stepping the protections that DEP offers. Owing to JIT engines' complexity, they often contain their own memory managers that rely on lower-level OS APIs. On some platforms, these APIs aren't subject to ASLR and thus don't supply random memory locations for emitted code. JIT engines emit machine code on the fly into memory and transfer execution to it as part of normal operation. This behavior is somewhat similar to how malicious memory safety exploits function. The properties of a JIT engine that make it so desirable for performance can run afoul of the memory protection mechanisms used to prevent the exploitation of memory safety vulnerabilities.

JIT engines can be both the source of a vulnerability and the means to exploit it. Security vulnerabilities are often introduced with any new sufficiently complex software component; JIT engines are no exception. They're notable, however, in pushing the traditional code compilation trust boundary to the user's desktop. Code compilation is nontrivial and includes the parsing and processing of untrusted data, byte code generation, and native-code emission.

Attacking JIT Engines

The first published exploit technique involving a JIT engine was the *JIT spray*. It took advantage of the Intel x86 processor's ability to reference data in the instruction stream by embedding immediate values directly into instructions. By using constant values in an interpreted language such as JavaScript, the exploit could influence the engine to include them directly in the emitted native code. So, if an attacker embedded instruction fragments in those values and redirected execution in the immediate data, the processor executed them. An attacker could place an entire attack payload into memory by inserting enough of those values into a higher-level script. This attack has since been ported to Firefox's and Opera's JavaScript engines.

Some JavaScript JIT engines allocate memory with readable, writable, and executable page permissions. These permissions aren't hardened after code emission, leaving them open to security vulnerabilities. Our research discovered that many pages allocated by a JIT engine were adjacent to regular heap memory. If an attacker can modify the native code written to these pages via a traditional buffer overflow or a 4-byte-write-anywhere vulnerability, he or she can achieve blind code execution. The term "blind code execution" stems from not needing to infer where in memory the JIT-emitted code resides—attackers need only a handle to the original high-level script. Because the original JavaScript functions are what's being overwritten, invoking the attack payload is as simple as calling a function by name. Both ASLR and DEP are largely ineffective against such an attack, and it's aided by allowing JavaScript to force the JIT engine into allocating many native-code pages.

Because JIT engines are often designed with custom memory allocators, developers must reimplement code page tracking through complex data structures. These structures often resemble doubly linked lists; when the list metadata is stored inline, they're often the target of attackers. This is especially true with the heap overflows that ran rampant through the early 2000s. Securely storing this sensitive information requires a separate out-of-band data structure or protection mechanisms such as heap cookies and list unlink checks. The NanoJIT assembler, used by both TraceMonkey and Adobe Flash Player, employs a doubly linked list named CodeList to track JIT page allocations. The metadata for this list resides at static offsets from the base of every JIT-allocated code page, not employing any of the protections of a modern heap implementation. Overwriting these pointers lets an attacker write 4 bytes of his or her choosing anywhere in memory once the allocator has reclaimed the page.

To circumvent the protection that DEP provides, recent memory safety exploits have used ROP (return-oriented programming) payloads. Because DEP doesn't let attackers inject new code into a process, ROP strings together pieces of existing code by creating an artificial program stack. Typically, ROP code segments (*gadgets*) are carved from existing code libraries. JIT engines introduce a new source of gadgets in the emitted code itself, giving attackers more influence. ROP becomes substantially harder when ASLR is in effect, because the attack must know all the code pieces' offsets. However, reusing JIT-emitted code is possible even when ASLR is in effect. Some JIT engines can be forced to emit duplicate pages of code throughout memory; this process is largely controlled or influenced by untrusted scripts. An attacker can force a JIT engine to allocate many executable code pages of memory, all containing the same predictable sequences of native-code instructions. Even with ASLR fully enforced, this code reuse attack is practical, not theoretical.

Unlike the code that shipped with your Web browser, JIT-emitted code is tied directly to existing object lifetimes. For example, when a browser loads a webpage, it might allocate many C++ objects to represent the various statements in the embedded JavaScript. The JIT-emitted native code might reference these objects directly, along with many other components in the JavaScript interpreter. Owing to the dynamic nature of the JavaScript typing system, these objects are routinely modified. JIT-emitted code is often optimized to operate on these objects with certain assumptions, such as their type and state when the code was produced. When these components become out of sync, the opportunity arises for what we call *incorrect JIT code emission*. Such bugs manifest themselves as a hybrid of traditional memory safety and logic vulnerabilities.

Hardening JIT Engines

It might seem impossible to sufficiently secure any collection of software that embeds a JIT engine. However, each compilation step that can lend itself to attack is also an opportunity to harden the application runtime, occasionally even surpassing an OS's own protections. Because JIT engines have total control over their output, you can take measures to prevent attackers from using them in exploits.

ASLR has proven its effectiveness by making it difficult for attackers to guess a specific memory mapping's location. You can duplicate this technique in the JIT output by inserting random NOP (no operation performed) instructions between native code elements, such as single instructions, basic blocks, and functions. This insertion will shift JIT-produced code in the page that has been allocated for it, making its reuse impractical. ASLR alone randomizes the page base address, leaving the bottom 10 bits constant on a 32-bit platform. Inserting NOP instructions adds entropy to the bottom bits. Similarly, shuffling the emitted functions' order can further obfuscate valid addresses.

Guard pages create a barrier between two adjacent memory allocations. You can reuse this idea to separate executable JIT code pages from normal read/write heap allocations. This prevents blind-execution attacks that employ a memory overwrite in an adjacent memory allocation from linearly copying attacker-controlled data into a JIT code page.

As we mentioned before, the original JIT spray attack exploited the Intel x86 architecture's ability to include data inline with instructions. The JIT engine can optionally reference these values from another location in memory at the expense of an extra instruction. Immediate values can also be obscured inline by a simple XOR encryption against a secret value known only to the JIT. This protection is called *constant blinding*.

To reuse JIT code or use a JIT spray, attackers must first influence the JIT engine to allocate a large number of memory pages. Real-world applications don't often require hundreds of Mbytes of emitted code. To prevent these attacks, you can restrict the upper limit of the memory the JIT engine can allocate, without negatively impacting legitimate applications.

The final protection mechanism is obvious: secure page permissions. Instead of marking JIT pages readable, writable, and executable, the JIT engine can mark them readable and executable only, once the emission is complete. Although effective, this mitigation can carry a performance penalty, chipping away at the very reason these engines exist. This is especially true when inline caching is used to speed up object property retrieval in dynamic languages such as JavaScript.

Any measurable performance decreases that these protections produce will be heavily weighed against their performance gains.

The attack surface that JIT engines introduce to client-side software can be intimidating. Even with the most advanced protection techniques in place, JIT engines still pose a risk. Incorrect JIT code emission vulnerabilities are only now becoming an issue, and this trend will likely continue.

When it comes to embedding in client-side software, JIT engines are still in their infancy. Only the latest JIT engines implement the most basic compiler concepts such as SSA (static single assignment), optimized register allocations, and dead code removal. JIT engines' role in client-side software, especially browsers, will certainly increase over time. JIT engines will begin appearing anywhere high performance of interpreted code is required; so will their accompanying security risks. ■

Chris Rohlf is an independent security consultant at Leaf SR. Contact him by visiting http://leafsr.com.

Yan Ivnitskiy is a senior security consultant for Matasano Security. Contact him at yan@matasano.com.

Have an idea for a future article? Email editors Dino A. Dai Zovi (ddz@theta44.org) and Alexander Sotirov (alex@sotirov.net).