\*Extended content for CVE-2022-28672

**JIT**

**Content:**

[**The Return of The JIT (Part 1) - ρ (rh0dev.github.io)**](https://rh0dev.github.io/blog/2017/the-return-of-the-jit/)

**GitHub:**

[**expdev/CVE-2017-5375\_ASM.JS\_JIT-Spray at master · rh0dev/expdev · GitHub**](https://github.com/rh0dev/expdev/tree/master/CVE-2017-5375_ASM.JS_JIT-Spray)

**The return of the JIT (part 1)**

Quá dài để đọc: đây là WU về **ASM.JS JIT-Spray** trong Mozilla Firefox (x86 32-bit) trên hệ điều hành Windows, biểu diễn thông qua 2 CVE: CVE-2017-5375 và CVE-2017-5400. Nó cho phép ta 100% vượt qua DEP và ASLR.

Lời tác giả:

“Tôi luôn thích ý tưởng của JIT-Spray kể từ khi lần đầu nhìn thấy nó được sử dụng cho [Flash](https://dl.packetstormsecurity.net/papers/shellcode/Writing-JIT-Spray-Shellcode.pdf) vào năm 2010. **JIT-Spray** đã được sử dụng để exploit bugs trong [Apple Safari](https://www.exploit-db.com/exploits/14221), tạo ra [info leak gadgets in Flash](http://zhodiac.hispahack.com/my-stuff/security/Flash_Jit_InfoLeak_Gadgets.pdf), tấn công rất nhiều phần mềm client khác, và đã từng bị lạm dụng [Microsoft’s WARP Shader JIT Engine](https://sites.google.com/site/bingsunsec/WARPJIT/JIT%20Spraying%20Never%20Dies%20-%20Bypass%20CFG%20By%20Leveraging%20WARP%20Shader%20JIT%20Spraying.pdf).

Ý tưởng này chưa bao giờ chết, và **JIT-Spray** đã quay trở lại.”

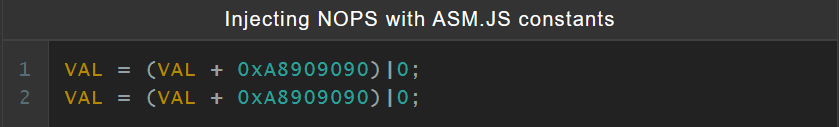
**JIT-Spray**

Nó đơn giản hóa việc exploit một memory corruption bug ví dụ như Use-After-Free, bởi vì attacker chỉ cần kiểm soát instruction poiter và nhảy tới JIT-Sprayed shellcode. **Không cần đến leak code locations hay base addresses của DLLs, và không cần bất kì code-reuse**.

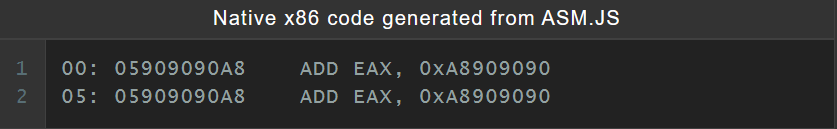
JIT-Spray thường khả thi khi có các điều kiện:

1. Machine code có thể giấu trong constants của ngôn ngữ lập trình bậc cao như JavaScript: Bypass được **DEP**.
2. Attacker có thể buộc JIT compiler phát hành (emit) các constants ra rất nhiều vùng code thực thi được (executable code regions) nơi mà các địa chỉ có thể đoán được: Bypass được **ASLR**.

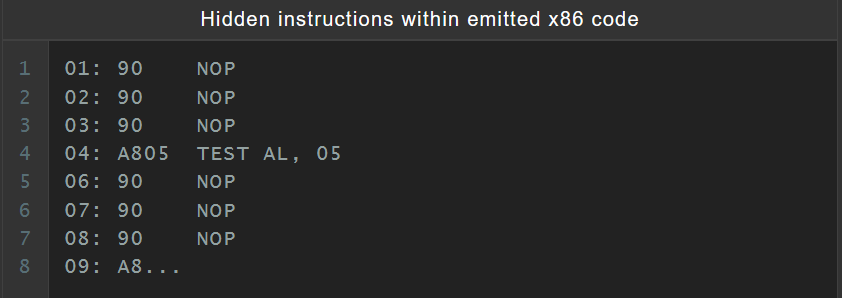
Ví dụ để đạt được điều kiện (1), chúng ta có thể chèn NOPS (0x90) trong code của **ASM.JS** với:



ASM.JS compiler của Firerox sẽ sinh ra đoạn code x86 sau đây:



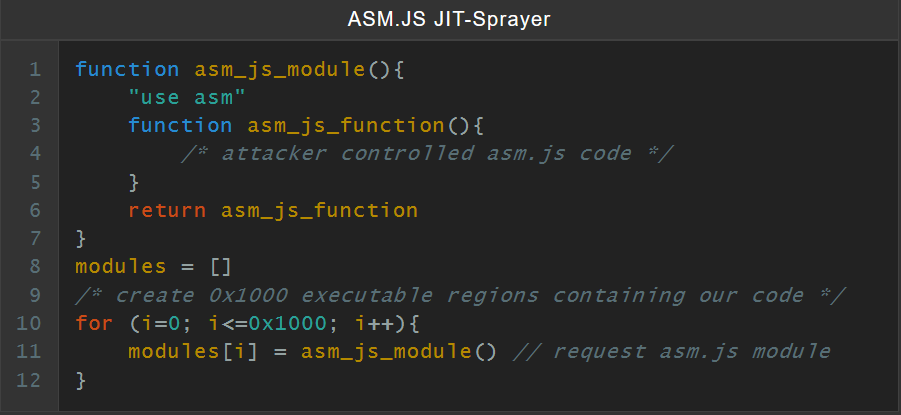
Khi chúng ta nhảy đến chỗ offset 01 (giữa instruction thứ nhất), chúng ta có thể execute hidden code:



Bởi vì máy sẽ đọc từng bytes bắt đầu ở vị trí ta nhảy đến để biến thành lệnh ASM, cho nên chỉ cần lệch 1 bytes cũng có thể biến thành code khác do flow đọc byte code (opcode) sẽ bị thay đổi.

Từ đó, ở trong 4-byte constants, chúng ta có 3 bytes để giấu code và 1 byte (0xA8) để gói lệnh **Add eax,…** thành các lệnh NOP (NOP-like instruction) và test **al, 0x05**.

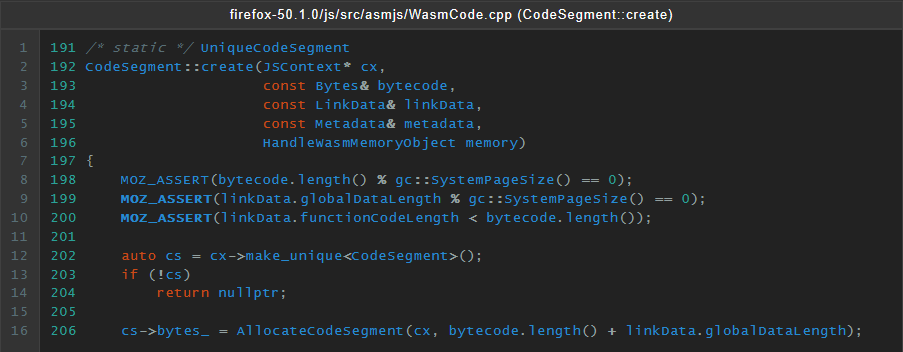
Để đạt được điều kiện (2) tức tạo ra rất nhiều vùng executable chứa code của chúng ta, chúng ta cần yêu cầu (gọi) module **ASM.JS** nhiều lần:



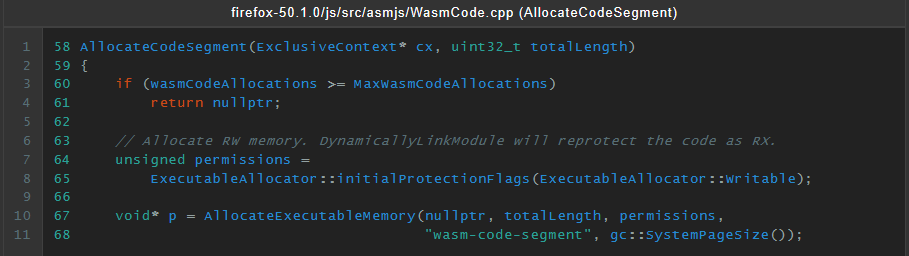
Về mặt kĩ thuật, **ASM.JS** là một **ahead-of-time (AOT)** compiler và **không phải là một just-in-time (JIT) compiler**. Cho nên hàm asm\_js\_function() không nhất thiết phải được gọi để đưa mã máy của ta vào một địa chỉ có thể đoán được (tức là tạo ra hàm đó là được, không nhất thiết phải gọi nó ở bất kì đâu). Và chỉ cần tải một trang web chứa tập lệnh **ASM.JS** là đủ.

**Lỗ hổng**

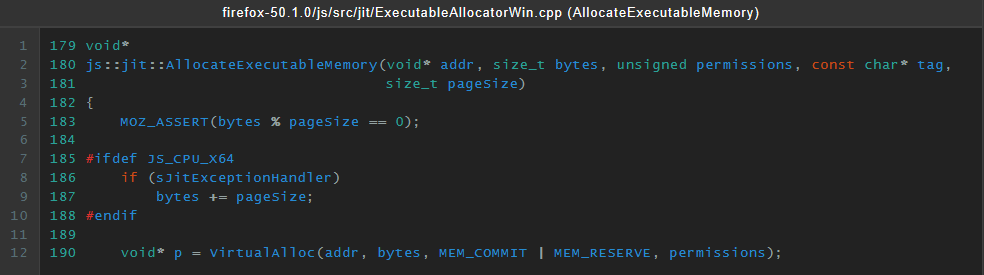
Với mỗi lần một module ASM.JS được gọi, **CodeSegment::create()** được gọi từ đó gọi cả **AllocateCodeSegment()** trong **firefox-50.1.0/js/src/asmjs/WasmCode.cpp** dòng #206 (dựa trên mã nguồn của [Firefox 50.1.0](https://ftp.mozilla.org/pub/firefox/releases/50.1.0/source/firefox-50.1.0.source.tar.xz)):



**AllocateCodeSegment()** còn gọi cả **AllocateExecutableMemory()** ở dòng #67:

****

Cuối cùng, **AllocateExecutableMemory()** gọi **VirtualAlloc()** , trả về một vùng RW (PAGE\_READWRITE) với kích thước bộ nhớ được cung cấp là 64KB (0xXXXX0000) (firefox-50.1.0/js/src/jit/ExecutableAllocatorWin.cpp, dòng #190).



Nếu ta đặt breakpoint tại VirtualAlloc() trong WinDbg, chúng ta có call stack tương ứng trong lúc chạy (Firefox 50.1.0):

0:000> kP a

# ChildEBP RetAddr

00 008fe060 670ef66e KERNEL32!VirtualAllocStub

01 (Inline) -------- xul!js::jit::AllocateExecutableMemory+0x10 [c:\builds\moz2\_slave\m-rel-w32-00000000000000000000\build\src\js\src\jit\executableallocatorwin.cpp @ 190]

02 008fe078 670f65c7 xul!AllocateCodeSegment(

class js::ExclusiveContext \* cx = 0x04516000,

unsigned int totalLength = <Value unavailable error>)+0x23 [c:\builds\moz2\_slave\m-rel-w32-00000000000000000000\build\src\js\src\asmjs\wasmcode.cpp @ 67]

03 008fe0b8 670de070 xul!js::wasm::CodeSegment::create(

struct JSContext \* cx = 0x04516000,

class mozilla::Vector<unsigned char,0,js::SystemAllocPolicy> \* bytecode = 0x08c61008,

struct js::wasm::LinkData \* linkData = 0x08c61020,

struct js::wasm::Metadata \* metadata = 0x06ab68d0,

class JS::Handle<js::WasmMemoryObject \*> memory = class JS::Handle<js::WasmMemoryObject \*>)+0x67 [c:\builds\moz2\_slave\m-rel-w32-00000000000000000000\build\src\js\src\asmjs\wasmcode.cpp @ 206]

04 008fe184 6705f99d xul!js::wasm::Module::instantiate(

struct JSContext \* cx = 0x04516000,

class JS::Handle<JS::GCVector<JSFunction \*,0,js::TempAllocPolicy> > funcImports = class JS::Handle<JS::GCVector<JSFunction \*,0,js::TempAllocPolicy> >,

class JS::Handle<js::WasmTableObject \*> tableImport = class JS::Handle<js::WasmTableObject \*>,

class JS::Handle<js::WasmMemoryObject \*> memoryImport = class JS::Handle<js::WasmMemoryObject \*>,

class mozilla::Vector<js::wasm::Val,0,js::SystemAllocPolicy> \* globalImports = 0x008fe200,

class JS::Handle<JSObject \*> instanceProto = class JS::Handle<JSObject \*>,

class JS::MutableHandle<js::WasmInstanceObject \*> instanceObj = class JS::MutableHandle<js::WasmInstanceObject \*>)+0x94 [c:\builds\moz2\_slave\m-rel-w32-00000000000000000000\build\src\js\src\asmjs\wasmmodule.cpp @ 689]

05 008fe260 6705aae6 xul!TryInstantiate(

struct JSContext \* cx = 0x04516000,

class JS::CallArgs args = class JS::CallArgs,

class js::wasm::Module \* module = 0x08c61000,

struct js::AsmJSMetadata \* metadata = 0x06ab68d0,

class JS::MutableHandle<js::WasmInstanceObject \*> instanceObj = class JS::MutableHandle<js::WasmInstanceObject \*>,

class JS::MutableHandle<JSObject \*> exportObj = class JS::MutableHandle<JSObject \*>)+0x1e6 [c:\builds\moz2\_slave\m-rel-w32-00000000000000000000\build\src\js\src\asmjs\asmjs.cpp @ 7894]

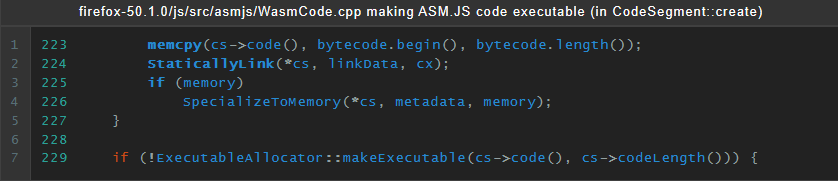
06 008fe2c4 35713638 xul!InstantiateAsmJS(

struct JSContext \* cx = 0x04516000,

unsigned int argc = 0,

class JS::Value \* vp = 0x008fe2f0)+0x88 [c:\builds\moz2\_slave\m-rel-w32-00000000000000000000\build\src\js\src\asmjs\asmjs.cpp @ 8008]

Sau khi quay trở về method **CodeSegment::create()**, compiled code của ASM.JS được copy tới vùng RW (firefox-50.1.0/js/src/asmjs/WasmCode.cpp, line #223), và ở line #229 vùng RW đó được cấp quyền thực thi (PAGE\_EXECUTE\_READ) với hàm **ExecutableAllocator::makeExecutable()** sẽ bao gồm cả **VirtualProtect()**.



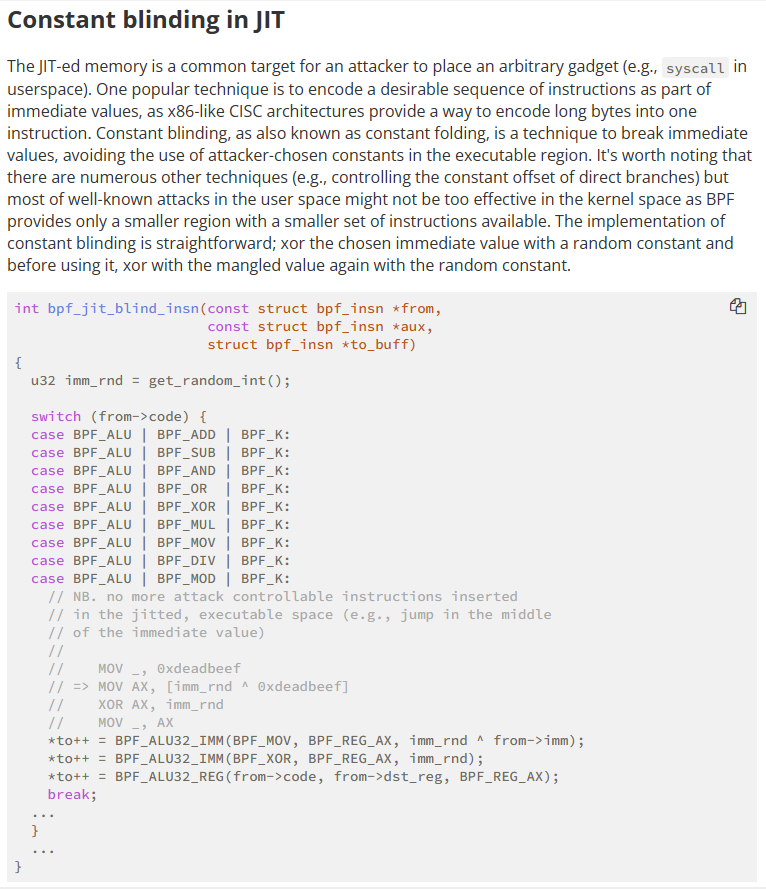
Sử dụng một module ASM.JS nhiều lần đồng nghĩa với việc tạo ra rất nhiều vùng RX. Do mức độ phân bổ chi tiết của VirtualAlloc (64KB) chúng ta có thể chọn một địa chỉ cố định, ví dụ như 0x1c1c0000) và có thể chắc chắn rằng machine code ta phát hành nằm tại đó (chưa payload ẩn của chúng ta).

Người đọc tinh ý có thể nhận ra **constant blinding** bị thiếu và cho phép phát hành constants của ASM.JS dưới dạng x86 code.

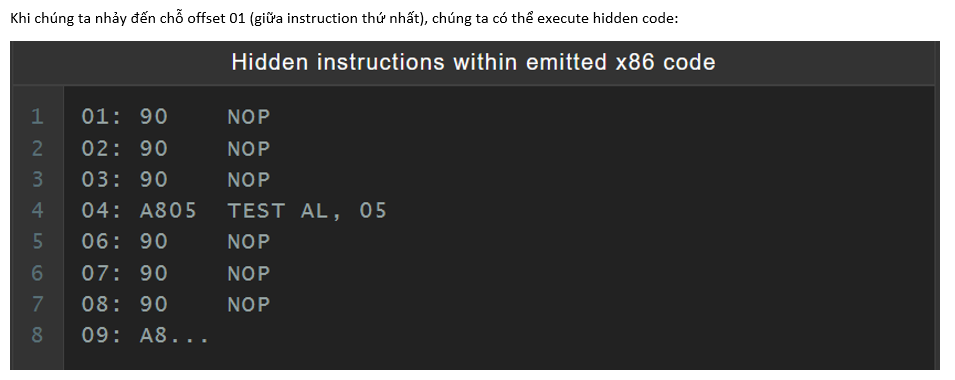
###########

Constant blinding:

[Hardening Hostile Code in eBPF - Analysis on Kernel Self-Protection: Understanding Security and Performance Implication (samsung.github.io)](https://samsung.github.io/kspp-study/bpf.html)



Có thể hiểu là **constant blinding** sẽ không cho phép nhảy vào giữa một immediate value như kiểu dưới đây:



Việc **không có** **constant blinding** trong chương trình sẽ giúp ta làm điều đó.

###########

**Show me a PoC!**

Hãy nhìn [proof of concept](https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/WinExec_cmd_Firefox_50.1.0.html) trông như nào trên thực tế: chúng ta giấu payload trong ASM.JS constants và request rất nhiều lần đến ASM.JS. Từ đó, ta phun rất nhiều vùng executable code để đạt được (một/những) địa chỉ có thể đoán trước được.

Payload chứa 2 phần:

1. Rất nhiều NOP-sled (dòng [#35 to #74](https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/WinExec_cmd_Firefox_50.1.0.html#L35)): để đến được nó, chúng ta có thể chọn một địa chỉ đoán trước được, ví dụ như 0x1c1c0053 và đặt EIP tới nó.
2. Shellcode (dòng [#75 to #152](https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/WinExec_cmd_Firefox_50.1.0.html#L75)): nó làm việc với **kernel32!WinExec()** và thực thi ***cmd.exe.***

Payload chỉ chứa lệnh dài nhất là 3-byte ngoại trừ các lệnh **MOV** – được handle kiểu khác. Nó được sinh tự động bằng một công cụ chuyển đổi tên là *shellcode2asmjs*, sử dụng Nasm assembler và Distorm3 diassembler. Payload được truyền cảm hứng mạnh bởi [Writing JIT-Spray-Shellcode](https://dl.packetstormsecurity.net/papers/shellcode/Writing-JIT-Spray-Shellcode.pdf).

Vì không có sự lạm dụng memory coruption nào ở trong cái PoC này, bạn phải phải đặt EIP ở debbuger khi được nhắc.

**Dynamic Payloads**

Above exploits contain “hardcoded” payloads within constants. That makes it kind of cumbersome to use different shellcodes. However, we can generate ASM.JS scripts on the fly and invoke them during runtime. A [PoC](https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/WinExec_cmd_Firefox_50.1.0_dynamic.html) where payloads are exchangeable uses the following:

* JavaScript code creates ASM.JS script-code dynamically. The ASM.JS script is included with the Blob JavaScript API (line [#88 to #137](https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/WinExec_cmd_Firefox_50.1.0_dynamic.html#L88)).
* A custom VirtualAlloc stage0. It allocates RWX pages and copies the actual stage1 payload (i.e. metasploit shellcode) to it. Afterwards, stage0 jumps to stage1 (line [#53 to #69](https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/WinExec_cmd_Firefox_50.1.0_dynamic.html#L53)).

This way, you can replace the payload with your favorite shellcode of choice (line [#33](https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/WinExec_cmd_Firefox_50.1.0_dynamic.html#L33)). The PoC and especially the stage0 payload were also auto-generated with the custom *shellcode2asmjs* tool.