Chromium WebAssembly 취약점 사례 분석: Overflow, Underflow 관련 사례를 중점으로

이재홍, 최형기
성균관대학교 소프트웨어융합대학
dymic9065@gmail.com, meosery@skku.edu

Analysis of Security Vulnerability Cases on Chromium WebAssembly: Focus on Cases Related to Overflow and Underflow

Jae-Hong Lee, Hyoung-Kee Choi
College of Computing and Informatics, Sungkyunkwan University

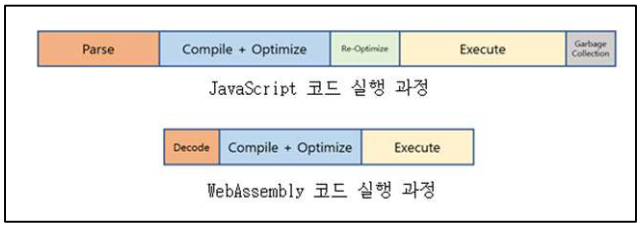
요 약

본 논문은 WebAssembly 가 도입된 2017 년부터 현재 2021 년까지 발생한 보안 취약점을 분석하고 분류하여, WebAssembly 에 대한 개발자들의 이해도를 높이고 WebAssembly 도입에 생길 수 있는 문제점들을 정리한다. 특히 CVE-2018-6092(Integer Overflow), CVE-2018-6036(Underflow) 사례들을 제공된 PoC 를 통하여 재현하고, PoC 코드, 원인 코드와 대처 코드까지 분석한다.

1. 서론

웹 플랫폼의 성숙으로 상호작용하는 3D 시각화, 오디오 및 비디오 소프트웨어, 게임과 같은 정교하고 까다로운 웹 응용 프로그램이 등장했고, 효율성과 보안성이 웹에서 매우 중요해졌다.[1]

이러한 시장의 빠른 속도와 안정성에 대한 요구에 2015 년에 WebAssembly(이하 Wasm)가 등장하였고, 2017 년 대부분의 브라우저에서 관련 기능을 제공하기 시작하였다. 2019 년 W3C(World Wide Web Consortium)에서는 HTML, CSS, JavaScript 에 이어 4 번째 언어로 Wasm 를 공식 권고하였다.[2]



(그림 1) JavaScript, WebAssembly 코드 실행 과정 비교

Wasm 은 어느 브라우저에서 C/C++ 나 비슷한 언어의 코드를 Native Speed 에 가깝게 실행할 수 있게 디자인된 Bytecode 언어이다. Wasm 은 컴파일된 코드를 사용하기 때문에, (그림 1)과 같이 JavaScript 보다 더 적은 과정으로, 더 빠른 실행속도를 갖는다. Wasm 은 독립된 가상머신 환경인 sandbox 환경에서 운영되기 때문에 보안성이 높다.[3]

본 논문에서는 Google Chrome, Microsoft Edge, Opera 등 2020 년 기준 도합 68%이상의 사용자를 보유한 브라우저들의 기반인 Chromium 에서 발생한 취약점들을 분석한다.[4] Wasm 도입 초기부터 2021 년까지 Wasm 에서 생긴 보안 취약점을 분석하여 유형을 분류하고, 특히 Data Type 관련된 CVE-2018-6092(Overflow), CVE-2018-6036(Underflow) 사례를 분석한다.

2. 용어 및 개념

Overflow 는 언어 자료형 설계 상의 크기보다 큰 연산을 처리하거나, 자료구조 설계 상의 크기를 넘어서는 행동을 할 때 발생하는 오류이다. 본 논문에서는 Integer 자료형의 최대치보다 큰 값의 연산을 처리하는데 발생하는 Integer Overflow 를 다룬다.

Underflow 는 Overflow 와 비슷하나, Overflow 가 특정 크기를 넘어서는 오류라면 Underflow 는 특정 크기보다 작은 값을 처리하면서 생기는 오류이다. 본 논문에서는 Integer 자료형의 최저치보다 작은 값의 연산을 처리하며 발생하는 Integer Underflow 를 다룬다.

Integer Overflow, Underflow 가 발생하게 된다면, 연산의 결과가 다르게 나오기 때문에 중요한 데이터에 접근하거나 예상과 다른 행동을 하는 등 다양한 문제를 일으킬 수 있다.

3. Chromium WebAssembly 보안 취약점 전체 분류

전체 Chromium 버그를 분석하기 위하여 Chromium Bug Report 에서 Wasm Bug-Security 중에서 Verified, Fixed 상태의 리포트를 대상으로 한다.[5]

전체 조사를 한 결과, 238 개의 리포트가 존재했으며, 그 중 202 개는 자세한 정보가 포함되지 않은 Chromium 내부 소프트웨어를 통한 리포트였다. 해당 리포트를 제외한 36 개 중의 15 개의 리포트가 CVE 에 등록되어 있는 리포트이다.

No.	CVE Number	Type	Publish Date	CVSS Scores
1	CVE-2018-17458	Out Of Bound	2019-01-09	6.8
2	CVE-2017-15401	Out Of Bound	2019-01-09	6.8
3	CVE-2017-5122	Out Of Bound	2017-10-27	6.8
4	CVE-2017-5116	Out Of Bound	2017-10-27	6.8
5	CVE-2017-5088	Out Of Bound	2017-10-27	6.8
6	CVE-2020-16015	Cache Reuse	2021-01-08	6.8
7	CVE-2018-6131	Use After Free	2019-06-27	6.8
8	CVE-2017-15399	Use After Free	2018-08-28	9.3
9	CVE-2018-6092	Overflow	2018-12-04	6.8
10	CVE-2018-6036	Underflow	2018-09-25	4.3
11	CVE-2017-5132	Data Type Confusion (Float32)	2018-02-07	6.8
12	CVE-2020-6381	Null Pointer Dereference	2020-02-11	6.8
13	CVE-2020-15994	Memory Corruption	2020-11-03	6.8
14	CVE-2018-6116	Out Of Memory	2018-12-04	4.3
15	CVE-2018-6061	Race Condition	2018-11-14	5.1

<표 1> WebAssembly 보안 취약점 전체 CVE 목록

<표 1>에 따르면, Out Of Bound 관련 리포트가 가장 많은 것을 확인할 수 있으며, 그 다음으로는 Use After Free 및 Cache Reuse 같은 메모리 재사용 관련 리포트가 가장 많았다. Overflow, Underflow, Data Type Confusion 같은 자료형 관련 취약점들이 그 다음으로 많았으며, 그 외에도 다양한 취약점들이 있었다.

결론적으로, Wasm 의 보안 취약점은 Memory 관련 오류들(OOB, UAF etc.)이 주를 이뤘으며, 다른 다양한 보안 취약점들도 확인되었다. 2017 년부터의 CVE 기록이 많지 않고, 큰 피해 사례가 없는 것으로 보아, Wasm 이 보안적으로 뛰어나다는 것을 알 수 있다.

Out Of Bound 보안 취약점 분석은 성균관대 이가현 저자의 논문에서 확인할 수 있으며, <표 1>에서의 3 번과 5 번에 해당하는 CVE-2017-5122 와 CVE-2017-5088 에 대한 분석을 확인할 수 있다.[6] 또한, Use After Free 보안 취약점 분석은 성균관대 현수빈 저자의 논문에서 <표 1>번의 8 번에 해당하는 CVE-2017-15399 에 대한 분석을 통해 확인할 수 있다.[7]

본 논문에서는 Data Type 관련 리포트 중, <표 1>번의 9 번과 10 번에 해당하는 CVE-2018-6092, CVE-2018-6036 에 대한 분석을 진행한다.

4. WebAssembly Module Structure

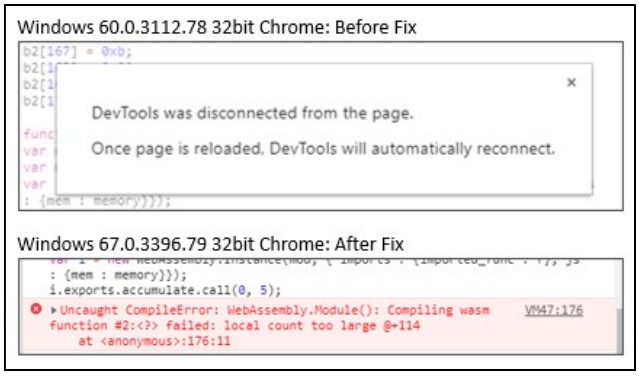
분석에 앞서 Wasm Module 구조에 대한 기본적인 이해가 필요하다. Module 은 처음 4byte 의 Magic Number 와 4byte 의 Version 정보로 시작한다. 그 후 여러 종류의 Section 들로 구성되어 있으며, 각 Section 은 `id`(Section Code)와 `payload_len`(Section Size)로 시작하여 종류마다 다른 구성을 갖는다. Section 들의 자세한 구조는 Wasm 공식 문서에서 확인할 수 있다.[8]

5. CVE-2018-6092

본 논문에서 CVE 분석은 Chromium Bug Report 에 공개된 Proof of Concept(이하 PoC)의 재현 및 분석, 문제 코드 및 수정 코드 분석 과정으로 이루어진다. CVE-2018-6092 는 Code Section 안의 Function Body 의 구성인 Local Entry 의 개수가 32bit 환경에서 일정 크기를 넘겼을 때, 관련 연산에서 발생하는 Integer Overflow 문제이다.

5.1. CVE-2018-6092 PoC 재현 및 분석

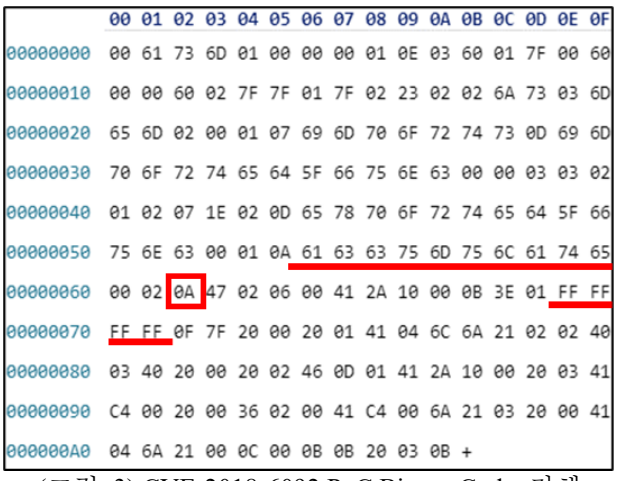
32bit 환경에서 발생하는 문제이므로, 재현은 Windows 60.0.3112.78 32bit Chrome 에서 진행하였다.



(그림 2) CVE-2018-6092 PoC 재현 Console Log

(그림 2)의 위의 결과를 보면, 코드 수정 전에는 PoC 코드를 실행 시 Chrome 에서 바로 Crash 를 일으켜 페이지가 정상 동작하지 않는 것을 확인할 수 있다. (그림 2) 아래 수정 후에는 Local Count 에 대한 오류를 정상적으로 일으키는 것을 확인할 수 있다.

Crash 가 발생하면 사용자의 민감한 정보 유출될 가능성이 있으며, 코드 실행이나 권환 확대에 대한 취약점 공격을 받을 수도 있다.



(그림 3) CVE-2018-6092 PoC Binary Code 전체

(그림 3) PoC 코드의 Wasm Module 의 이진코드에서 핵심적인 부분은 Local Entry 의 크기를 크게 선언하는 부분이기 때문에 해당 코드를 확인해야 한다. Local Entry 는 Code Section 에만 존재하기 때문에 해당 부분을 확인해야한다. id(Section Code)중에서 Code Section 에 해당하는 0x0A 를 0x62 위치에서 확인할 수 있으며(박스부분), 그 전 줄인 0x50 줄(밑줄 부분)을 보면, ASCII Code 로 `accumulate` 이라는 이름으로 함수가 선언되어 있다. 해당 Code Section 중에서 Integer Overflow 를 일으키는 0xFF 부분의 Code Section 코드의 앞부분을 분석하면 (그림 4)과 같다.

```
- section structure
0x0a    code section
0x47    payload length: 71
  - code section
0x02    function body counts:  2
    - function body 1
0x06    body size: 6
0x00    local count: 0
0x41    ┐
0x2A    │
0x10    │   function body
0x00    │
0x0B    ┘
    - function body 2
0x3E    body size: 62
0x01    local entry count: 1
      - Local entry
0xff    ┐
0xff    │
0xff    │    local count
0xff    │
0x0f    ┘
0x7f    local entry type: i32
```

(그림 4) CVE-2018-6092 PoC Code Section

(그림 4)에서 Code Section 은 두 개의 함수 코드를 포함하고 있으며, 14 번째 줄 이후 Function Body 2 에서 Local Entry 의 개수(박스 부분)는 32bit 으로 설계되어 있는데, 이 부분을 최대(0xfffffff0f)로 하여 Local Entry 개수와 관련된 연산에서 Integer Overflow 가 일어나기 쉽도록 유도하였다.

```
//...
b2[170] = 0xb;
function f(){print("in f");}
var memory = new WebAssembly.Memory(
    {initial:1, maximum:1});
var mod = new WebAssembly.Module(b2);
var i = new WebAssembly.Instance(
    mod,
    { imports : {imported_func : f},
    js : {mem : memory}}
);
i.exports.accumulate.call(0, 5);
```

(그림 5) CVE-2018-6092 PoC JavaScript

(그림 5)를 보면, 앞부분을 Binary 코드로 Buffer 를 채운 후에 6 번째 줄에서 Wasm Module 을 선언하고 Instance 를 생성하고, 12 번째 줄에서 앞서 보았던 오류 성 함수를 호출하는 것으로 PoC 가 마무리된다. Chrome 브라우저에서 코드를 실행시킬 시에 버전에 따라 (그림 2)과 같은 결과를 확인할 수 있다.

3.2 CVE-2018-6092 문제 코드 및 수정 코드

Chromium 에서 Integer Overflow 문제에 취약한 코드와 수정 코드는 (그림 6)과 같다.

```
if((count + type_list->size()) > kV8MaxWasmFunctionLocals){
    decoder->error(decoder->pc() - 1, "local count too large");
    return false;
}
```

```
DCHECK_LE(type_list->size(), kV8MaxWasmFunctionLocals);
if(count > kV8MaxWasmFunctionLocals - type_list->size()){
    decoder->error(decoder->pc() - 1, "local count too large");
    return false;
}
```

(그림 6) function-body-decoder-impl.h 취약점 수정 전(위), 후(아래)

만약 `count + type_list->size()` 연산의 결과가 integer 32bit 의 최대값(2^{31}-1)보다 크다면, (그림 6)의 위의 첫 번째 줄의 조건문에서 Integer Overflow 가 발생하게 되고 이것은 바로 Crash 로 이어진다.

대처로는, `DCHECK_LE()`를 통하여 `type_list->size()`가 `kV8MaxWasmFunctionLocals` 보다 작은 것을 확인한 후에, 기존의 더하기 방식의 조건문에서 빼기 방식의 조건문으로 변경하였다.

6. CVE-2018-6036

CVE-2018-6036 은 Wasm Module 을 제작할 때 Custom Section 에서 유효하지 않은 Section Length 에도 불구하고 Module 이 만들어지면서 생기는 문제이다.

6.1. CVE-2018-6036 PoC 재현 및 분석

재현은 취약점 수정 전의 버전인 Linux 61.0.3163.79 64bit Chrome 환경에서 진행하였다. 큰 Section Length 를 설정해야 하기 때문에 재현은 Buffer 크기를 제어하는 Main Thread 에서 불가능하며 Worker Thread 를 통해서 재현이 가능하다.

따라서 재현을 위하여 로컬서버를 구동하고, JavaScript 코드(index.html)를 통하여 PoC 코드가 포함되어 있는 코드(worker.js)를 Worker 로 지정하여 재현을 진행하였다.

```
▼Uncaught RangeError: WebAssembly.Module.customSections(): out of memory allocating custom section data
    (anonymous) @ worker.js:22
>
```

(그림 7) CVE-2018-6036 수정 이전 재현 Chrome DevTools Console Error Log

재현 결과, (그림 7)에서 확인할 수 있듯이, Worker Thread 코드인 worker.js 에서 문제를 일으키고 있다. 브라우저의 Main Thread 에서 일어난 이슈가 아니라 Worker Thread 에서 일어난 문제이기 때문에 직접적인 Crash 를 내진 않았지만, Worker Thread 에 Out Of Memory 오류를 보여준다.

```
1   var string_len = 0x0ffffff0 - 18;
2   var backing = new ArrayBuffer(string_len + 18);
3
4   // Binary code
5   var buffer = new Uint8Array(backing);
6   // Fill Buffer with bunch of "A"
7   buffer.fill(0x41);
8   // Wasm Magic
9   buffer.set([0x00, 0x61, 0x73, 0x6D], 0);
10  // Wasm Version
11  buffer.set([0x01, 0x00, 0x00, 0x00], 4);
12  // Section Define = 0 = custom section
13  buffer.set([0], 8);
14  // Section Length = 0
15  buffer.set([0x80, 0x80, 0x80, 0x80, 0x00],  9);
16  // Name Length = 0x0FFFFFF0 - 18
17  buffer.set([0xDE, 0xFF, 0xFF, 0x7F], 14);
18
19  var m = new WebAssembly.Module(buffer);
20  var c = WebAssembly.Module.customSections(
21      m, "A".repeat(string_len ));
```

(그림 8) CVE-2018-6036 PoC 코드 전체

(그림 8)의 PoC 코드를 분석하면, ArrayBuffer 는 0x0ffffff0 만큼의 크기를 가지고 구성이 되어있다. 앞의 18 Byte 는 Wasm Magic, Wasm Version, Section Code, Section Length, Name Length 로 채워져 있고, 뒤에는 Section Name 으로 0x41('A')로 모든 Buffer 가 채워져 있다. 15 번째 줄에서 확인할 수 있듯이, Section Length 에 포함되는 Section Name 의 크기가 0x0ffffff0-18 로 큼에도 불구하고 Section Length 는 0 으로 정의되었다.

6.2. CVE-2018-6036 문제 코드 및 수정 코드

```
1   while (decoder.more()) {
2   byte section_code = decoder.consume_u8("section code");
3   uint32_t section_length=decoder.consume_u32v("section length");
4   uint32_t section_start = decoder.pc_offset();
5   ...
6   uint32_t name_length = decoder.consume_u32v("name length");
7   uint32_t name_offset = decoder.pc_offset();
8   decoder.consume_bytes(name_length, "section name");
9   uint32_t payload_offset = decoder.pc_offset();
10  uint32_t payload_length = (
11      section_length - (payload_offset - section_start));
12  decoder.consume_bytes(payload_length);
```

(그림 9) module-decoder.cc 취약점 수정 전

(그림 8)의 Buffer 가 입력된다면 (그림 9)의 코드에서 문제를 일으킨다. `Section Code`, `Section Length`, `Name Length` 를 Buffer 로부터 받고 10 번째 줄에서 `payload_length` 를 선언하는데 `section_length` 를 사용한다. 하지만 앞서 (그림 7)처럼 `section_length` 를 0 과 같은 유효하지 않은 값을 넣는다면, `payload_length` 에 Underflow 가 발생한다. 이것은 이후에 12 번째 줄의 `decoder.comsume_bytes` 에서 유효하지 않은 `paload_length` 로 인해 Out Of Memory 를 일으킬 수 있다.

```
1   if(section_length < (payload_offset - section_start)){
2       decoder.error("invalid section length");
3       break;
4   }
```

(그림 10) module-decoder.cc 취약점 수정 후 코드

수정은 문제가 되는 (그림 9)의 10 번째 줄의 앞에서 (그림 10)와 같은 코드로 `section_length` 가 유효한 크기를 갖는지 확인하는 것으로 취약점을 수정하였다.

7. 결론

본 논문에서는 Chromium 에 WebAssembly 가 도입된 2017 년부터 2021 년까지의 모든 WebAssembly 보안 취약점을 확인하고 분류하였다. 분석 결과, WebAssembly 에서 발생하는 보안 취약점은 Memory 관련 오류들(OOB, UAF etc.)이 주를 이루었으며, 다른 오류들도 나타났다. 또한 처음 도입된 2017 년부터의 기록임에도 불구하고 15 개의 CVE 만이 존재하고 큰 피해 사례가 없는 것으로 보아, WebAssembly 가 보안적으로 뛰어나다는 것을 알 수 있다.

CVE-2018-6096(Overflow), CVE-2018-6036(Underflow) PoC 코드를 분석하고 원인 코드와 수정 코드를 분석한 결과, 분석한 두 사례 모두 입력 값에 대한 기본 연산에서 발생한 문제였으며, 따라서 이와 같은 입력 값에 대한 연산처리 시에 주의가 요구된다.

8. Acknowledgement

이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2020R1A2C1012708).

참고문헌

[1] Haas, Andreas & Rossberg, Andreas & Schuff, Derek & Titzer, Ben & Holman, Michael & Gohman, Dan & Wagner, Luke & Zakai, Alon & Bastien, Jf, Bringing the web up to speed with WebAssembly, 1, 2017

[2] Amy van der Hiel, World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation, W3C, https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en, 2019

[3] WebAssembly Community Group, Security, https://webassembly.github.io/spec/core/binary/, 2017

[4] StatCounter, 2020 Browser Market Share Worldwide, https://gs.statcounter.com/browser-market-share/all/worldwide/2020, 2021

[5] Google, Chromium WebAssembly Security Bugs, https://bugs.chromium.org/p/chromium/issues/list?q=Type%3DBug-Security%20-component%3ABlink%3EJavaScript%3EWebAssembly

[6] 이가현, WebAssembly 기능 도입으로 인해 크롬에서 발생한 오류들의 분석, 한국소프트웨어종합학술대회, 온라인, 2020

[7] 현수빈, 웹어셈블리 UAF(Use-After-Free) 취약점 사례 분석, 한국통신학회 추계종합학술발표회, 온라인, 2020

[8] WebAssembly Community Group, Binary Format, https://webassembly.github.io/spec/core/binary/, 2017