Reliable Mitigation of DOM-based XSS

Tobias Mueller

2014-09-07

about:me

- MSc. cand. Dipl. Inf.
- presenting results of diploma thesis / USENIX paper
- $\sim$ 45 min. presentation
- ask immediately, Q&A afterwards

Spoiler ▼

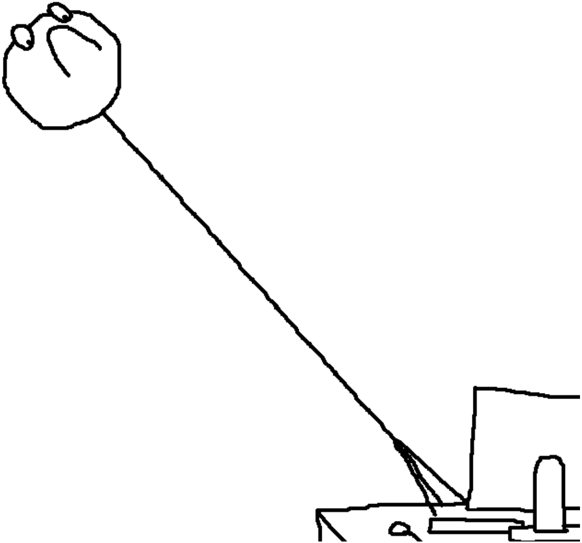
Is it possible to reliably defend against DOM-based XSS without breaking the Web?

- recognise new code from attacker-provided strings
- modified V8's scanner, WebKit's strings, and the bindings for Chromium
- evaluated protection, compatibility, and speed

→ yes, with some exceptions

Motivation

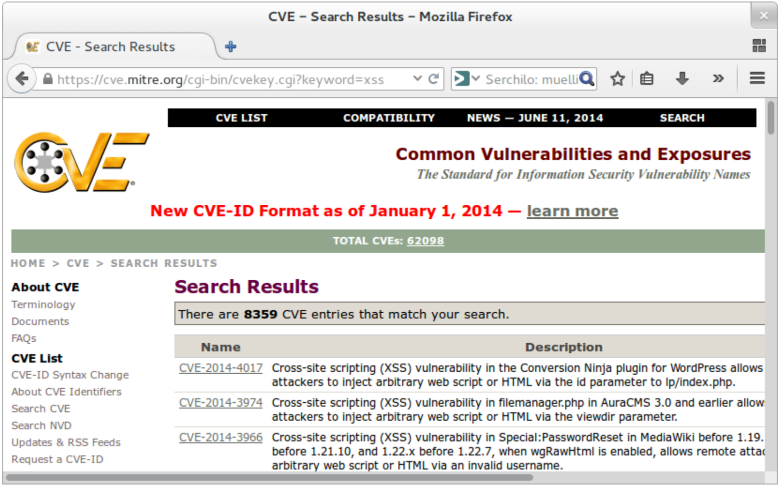
10.000 feet



Severity of XSS

- 2004: OWASP Top 4
- 2007: OWASP Top 1
- 2010: OWASP Top 2
- 2013: OWASP Top 3

Severity of XSS - 10% of CVEs are XSS

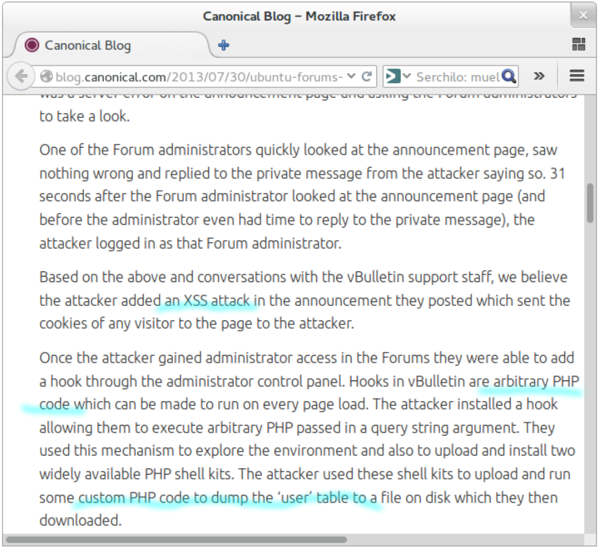


Severity of XSS

XSS is *very* common

and dangerous

Severity of XSS - 2 mio user records



to take a look.

One of the Forum administrators quickly looked at the announcement page, saw nothing wrong and replied to the private message from the attacker saying so. 31 seconds after the Forum administrator looked at the announcement page (and before the administrator even had time to reply to the private message), the attacker logged in as that Forum administrator.

Based on the above and conversations with the vBulletin support staff, we believe the attacker added an XSS attack in the announcement they posted which sent the cookies of any visitor to the page to the attacker.

Once the attacker gained administrator access in the Forums they were able to add a hook through the administrator control panel. Hooks in vBulletin are arbitrary PHP code which can be made to run on every page load. The attacker installed a hook allowing them to execute arbitrary PHP passed in a query string argument. They used this mechanism to explore the environment and also to upload and install two widely available PHP shell kits. The attacker used these shell kits to upload and run some custom PHP code to dump the 'user' table to a file on disk which they then downloaded.

Overview of section 2

Cross-Site Scripting (XSS)

Code Execution in the victim's browser

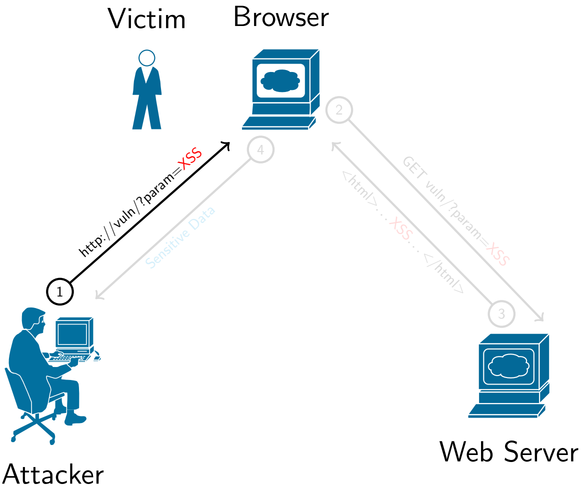
- (JavaScript) Code execution
- use all browser APIs
- use Web app in the name of the user
- obtain credentials
- spy on behaviour

Reflected XSS

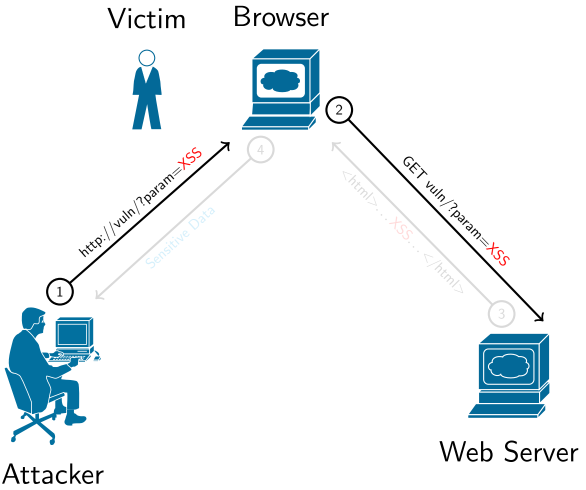
```php
<?php
// returning unsanitised data
echo $_GET['bar'];
?>
```

Attack: `http://foo/?bar=<script>alert("xss")</script>`

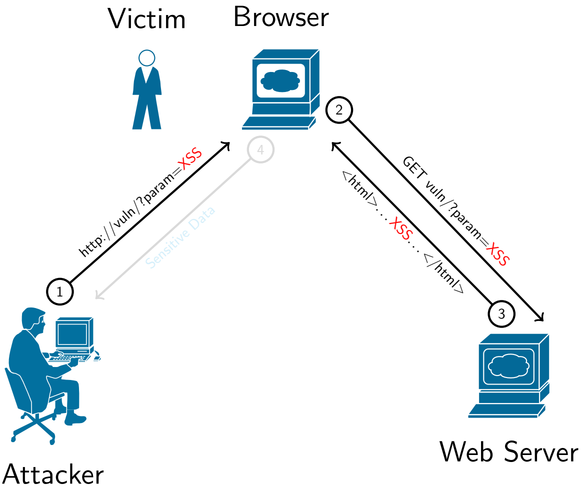
Reflected XSS



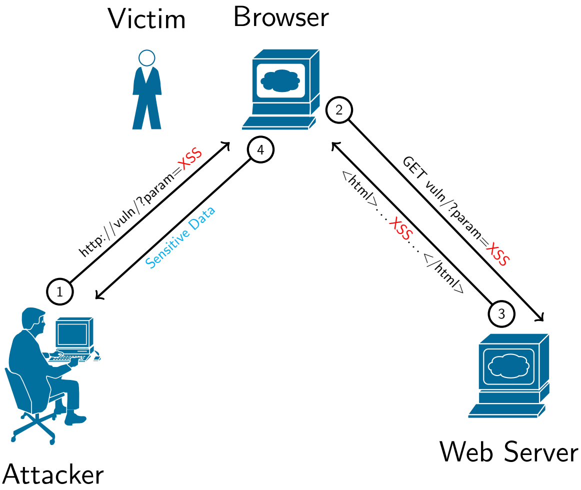
Reflected XSS



Reflected XSS



Reflected XSS



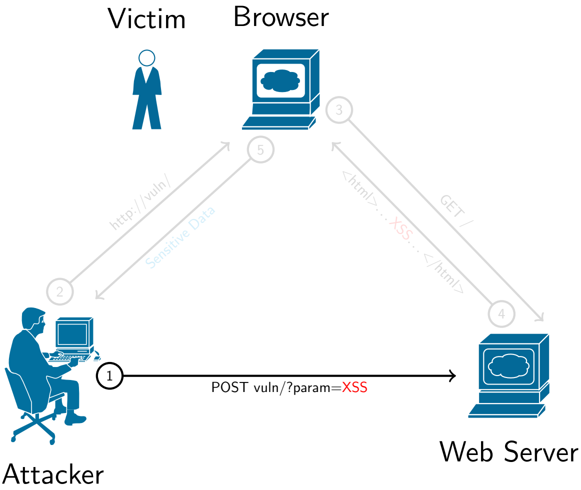
Stored XSS

```
<?php // store.php
store_in_db ('some_key', $_POST['bar']);
?>

<?php // retrieve.php
// returning unsanitised data
echo get_from_db ('some_key');
?>
```

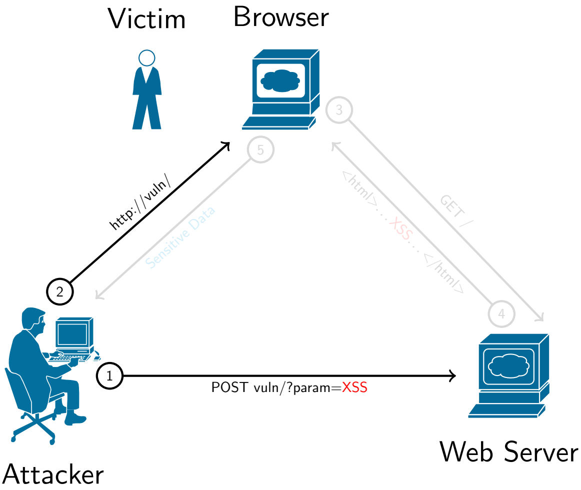
Attack:

1. POST `http://foo/?bar=<script>alert(1)</script>`
2. `http://foo/retrieve.php`

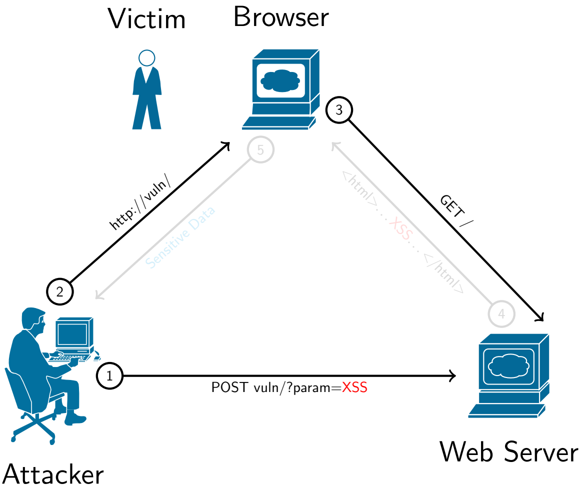
Stored XSS


Victim
Browser
3
5
http://vuln/
Sensitive Data
<html>...XSS...</html>
GET /
2
4
1
POST vuln/?param=XSS
Attacker
Web Server

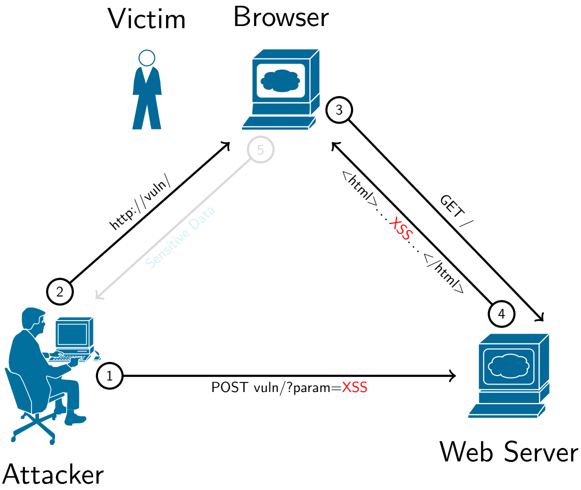
Stored XSS


Victim
Browser
3
5
http://vuln/
Sensitive Data
<html>...XSS...</html>
GET /
2
4
1
POST vuln/?param=XSS
Attacker
Web Server

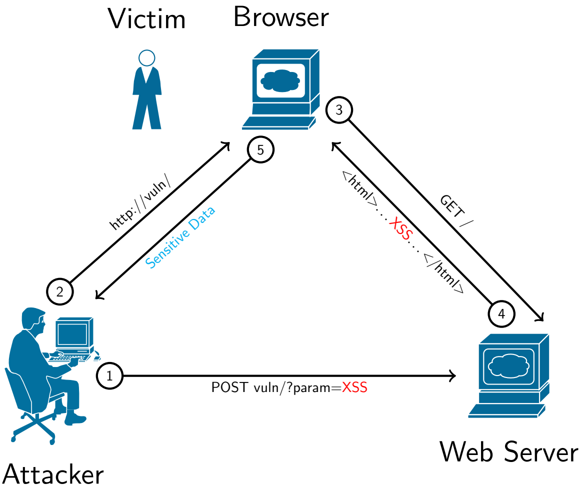
Stored XSS


Victim
Browser
3
GET /
5
http://vuln/
Sensitive Data
<html>...XSS...</html>
2
4
1
POST vuln/?param=XSS
Attacker
Web Server

Stored XSS


Victim
Browser
3
GET /
5
http://vuln/
Sensitive Data
<html>...XSS...</html>
2
4
1
POST vuln/?param=XSS
Web Server
Attacker

Stored XSS


Victim
Browser
3
GET /
5
http://vuln/
Sensitive Data
2
<html>. . . XSS. . . </html>
4
1
POST vuln/?param=XSS
Attacker
Web Server

DOM-based XSS

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos = document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system...
</HTML>
```

Attack:

```
http://vuln/welcome.html#name=<script>alert(1)</script>
```

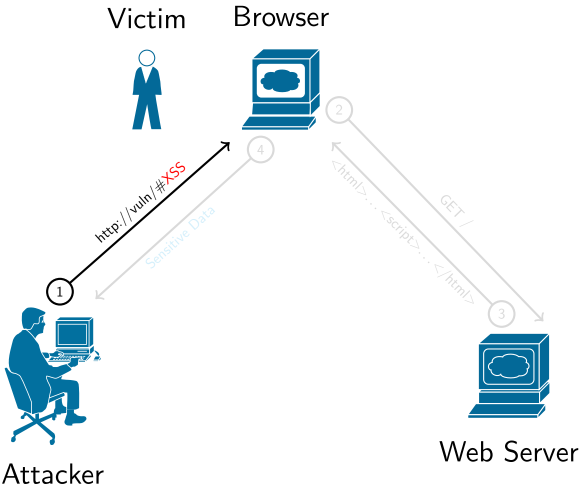
DOM-based XSS

- Neither Stored- nor Reflected-XSS !!111elfeins
- Client-side vulnerability
- Read from (attacker controlled) properties of the loaded document
  - `document.location`, `window.name`, etc...
- Write to security sensitive sinks
  - `eval`, `document.write`, etc...

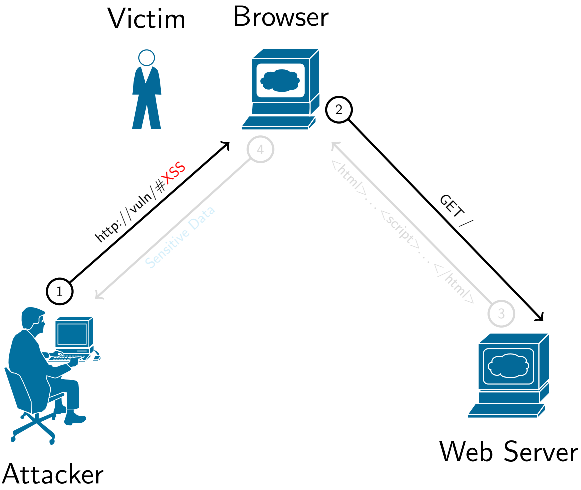
eval(document.location.hash.substring(1))

`http://lolcathost:8000/#alert(1)`

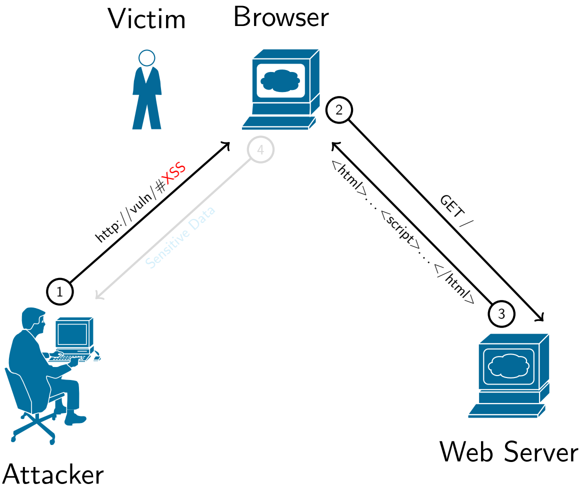
DOM-based XSS



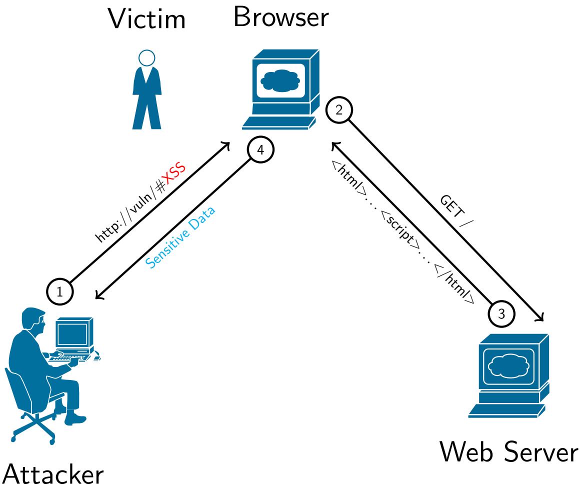
DOM-based XSS



DOM-based XSS



DOM-based XSS



DOM-XSS protection mechanisms

- server-side solutions
  - inappropriate as data does not leave the client
- turn off JavaScript ...
  - breaks the Web
- WebKit's XSS Auditor
  - "Only" smarter string matching
  - inherent weaknesses, e.g. in WebKit, not V8 → eval
- Block tainted JavaScript code
  - too coarse grained, breaks the Web:
  - `var name=d.URL.substring(d.URL.indexOf("name="))`

→ use knowledge of data flows to only allow data values and forbid code

Interlude: Recap

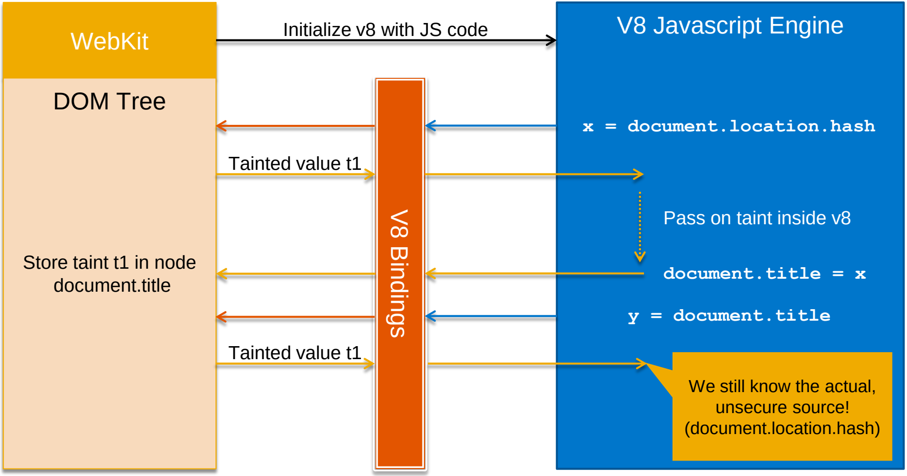
- XSS is a problem
- DOM-XSS is a client-side problem
- The client is the appropriate place for a fix
- The idea is to observe data flows to allow literals but block new code

Overview of section 3

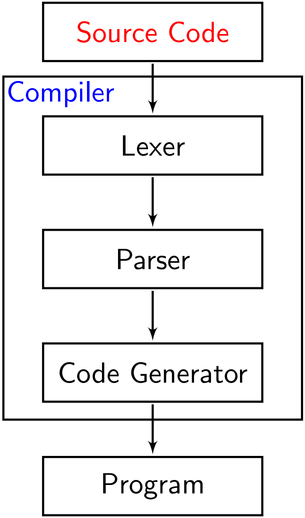
Taint Tracking

- Annotate data and track it throughout
- `perl -T`
- `navigator.taintEnabled()`

Automated data flow detection

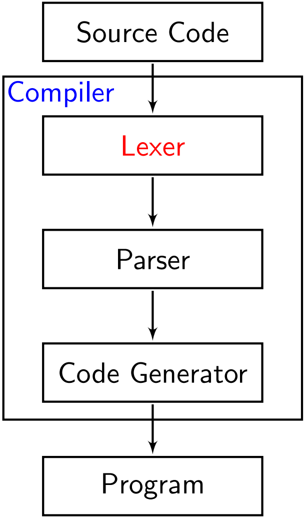


Compilation



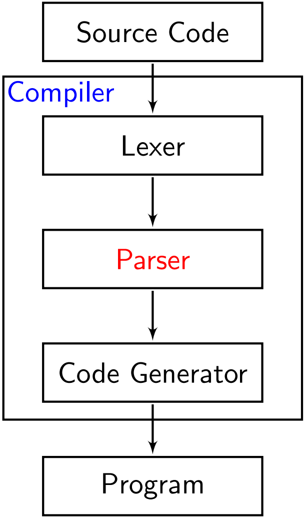
var foo = “*bar*” ;

Compilation



var	foo	=	"*bar*"	;
VAR	ID	EQ	STR	SEMI

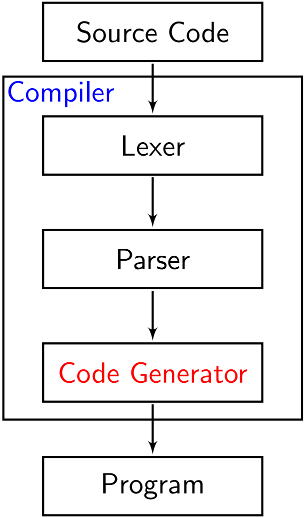
Compilation



var	foo	=	"*bar*"	;
VAR	ID	EQ	STR	SEMI

VariableStatement:
  var *VariableDeclarationList*

Compilation



var	foo	=	“*bar*”	;
VAR	ID	EQ	STR	SEMI

VariableStatement:
  var *VariableDeclarationList*

```
sub esp, 4
```

Architecture

Chromium

∼ 12,000k SLoC

WebKit	V8
HTML Rendering	JavaScript Execution
∼ 800k SLoC C++	∼ 700k SLoC C++

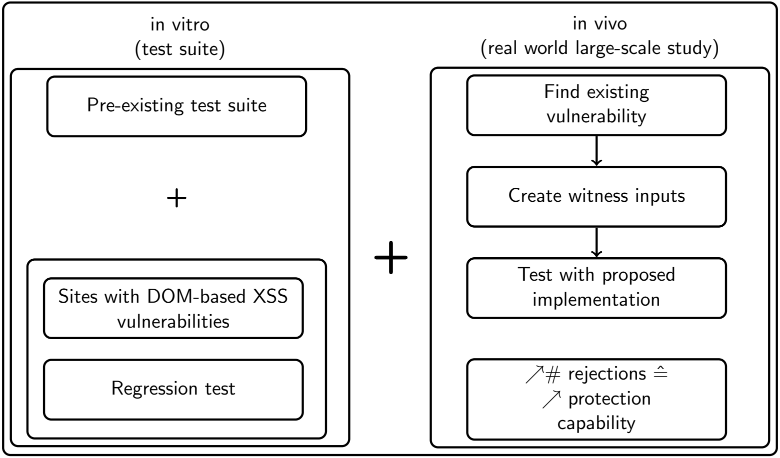
```
 * If it doesn't look good, you will get a pointer to the second container
 * back. You may preload that container with an ILLEGAL token.
 * The reason for that design is a bit wacky: I believe it more safe
 * as the container live on the stack of the caller. So they won't get
 * tampered with if they were on the callee's stack and some other functions
 * run in between. Although I have no data to back that up.
 */
Token::Container* Scanner::CheckTaint(Token::Container& current_container,
        Token::Container& illegal_container) {
    Token::Container* return_container_p;
    const Token::Value current_token = current_container.value();
    const bool is_tainted = current_container.is_tainted();

    if (is_tainted) {
        OS::Print("Tainted Token in scanner!!1 %s (%d)\n",
                Token::String(current_token), is_tainted);
        // We check the token's value and decide whether to allow or not
        switch (current_token) {
        case Token::STRING:
        case Token::TRUE_LITERAL:
        case Token::FALSE_LITERAL:
        case Token::NUMBER:
        // It may be useful to allow this to go through untaintedly.
        // We cannot call Token::String(EOS) and we prevent to get in
        // trouble if we wanted to report that token.
        case Token::EOS:
            // We have only so many tokens that we want to allow for now.
            return_container_p = &current_container;
            break;
        default:
            // All others we are replacing with an illegal token.
            return_container_p = &illegal_container;
            break;
        }
    } else {
        return_container_p = &current_container;
    }
    return return_container_p;
}
```

Overview of section 4

Evaluation

- Protection: Test cases and vulnerable top 10000 Web apps ($\sim$ 8% vuln.)
- Compatibility: Test cases and top 10000
- Execution speed: standard benchmarks against baseline

Protection - Setup

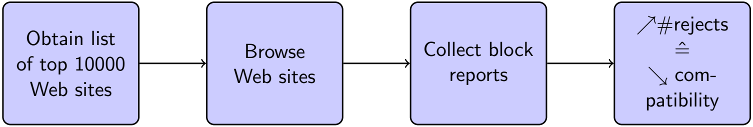


Protection - Results

	Without	XSS Auditor	Taint Aware browser
Exploitable Domains	757	545	0
Protection Rate	0%	28.01%	100%

Table : Protection Capabilities of the XSS Auditor and the taint browser

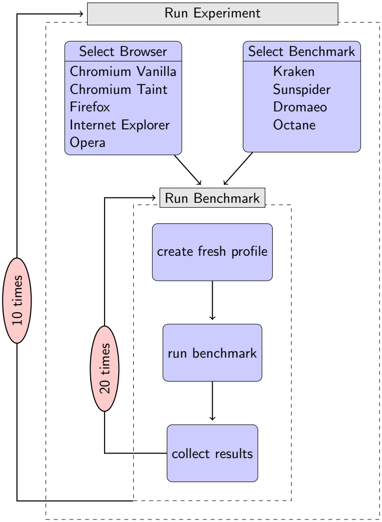
Compatibility - Setup



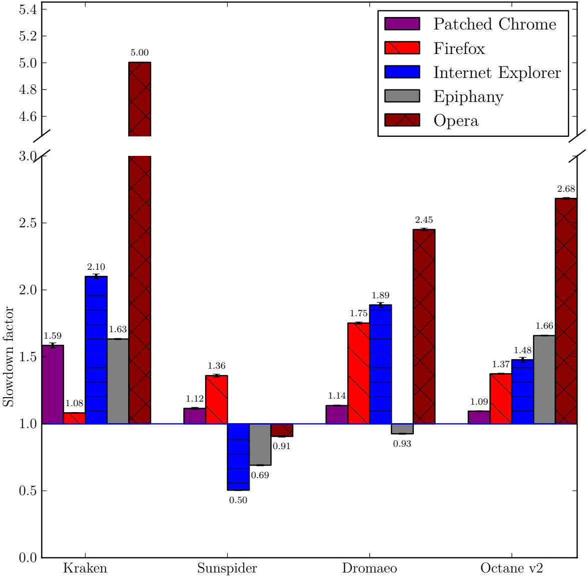
Compatibility - Results

8 (out of 10000) wrongfully blocked Web apps:
al.com, blogger.com, elpais.com, google.com, ixian.cn, miami.com, mlive.com, toyota.jp

Execution Speed - Setup



Execution Speed - Results - 23% vs. 39%, 49%, and 63%



Overview of section 5

Q&A

- Client-side protection mechanism against DOM-XSS
- Thorough evaluation of the proposed implementation
- Review of existing XSS protection mechanisms

to be read in
"**Precise Client-side Protection against DOM-based XSS**",
in: *23rd USENIX Security Symposium (USENIX Security 14)*.

Questions?