IS XSS SOLVABLE?

Don Ankney • LayerOne 2009

DISCLAIMER

This presentation represents personal work and has not been approved or vetted by Microsoft. I am solely responsible for its content.

OUTLINE

- XSS as a generic injection attack.
- What makes XSS unique, why it's important.
- Defending against XSS in general.
- Web Application Defense
  - One effective architecture design pattern.
  - Tool to help.
- Where to go from here.

DEFINING THE PROBLEM

INJECTION FLAWS

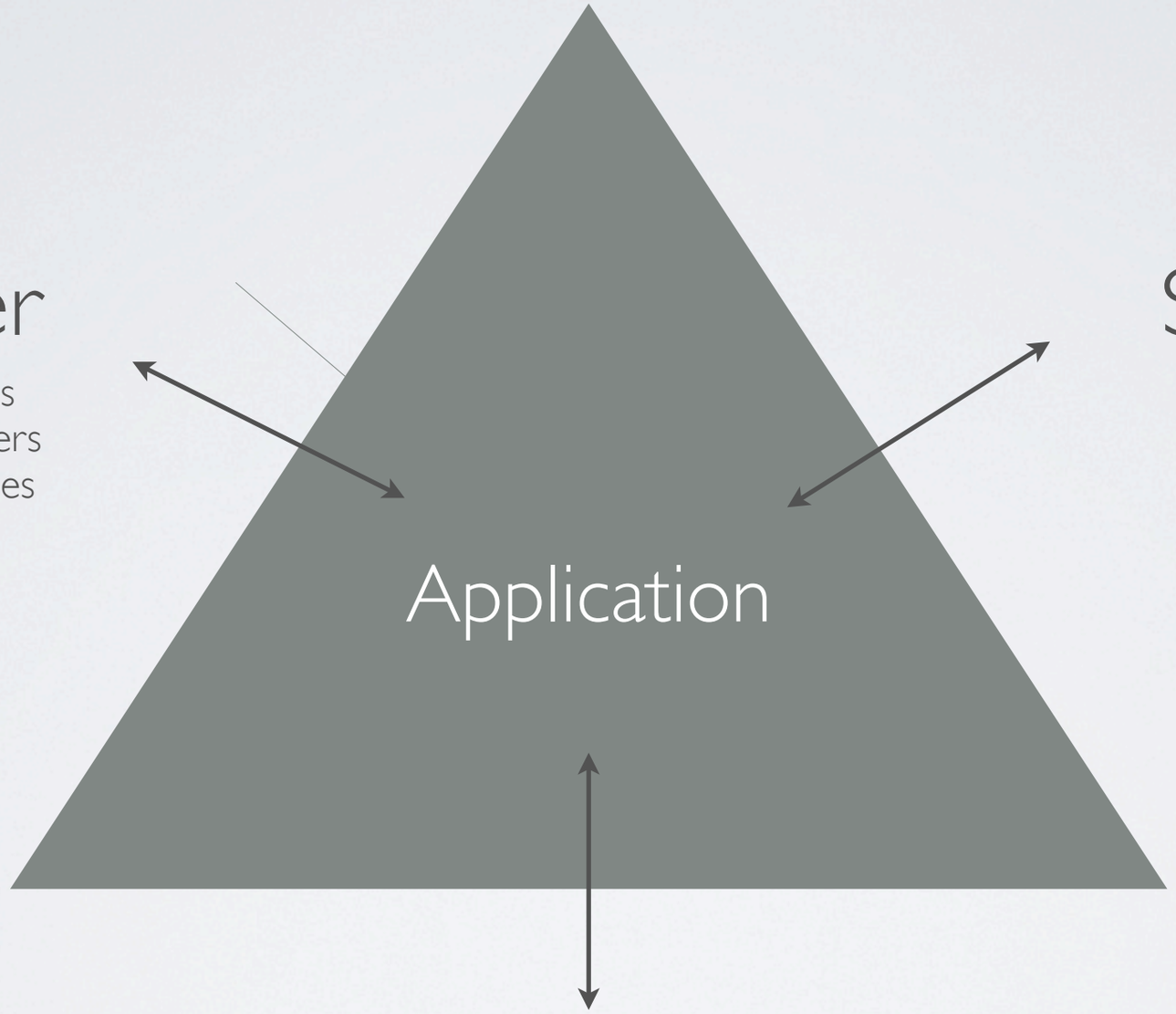
- Any interpreted code can be injected: LDAP, XML, HTTP, XAML, PHP, Python, Ruby, and most infamously, SQL.
- XSS is simply another form of interpreter injection, usually using Javascript.
- We can solve it just like we would solve any other injection problem*.

* except SQL injection -- we have a better solution there.

PREVENTING INJECTION

- Always keep track of your trust boundaries.
- Validate/sanitize your inputs.
- Properly encode your outputs.
- Use white lists, not black lists.
- Exercise the principal of least privilege.
- Validate your assumptions.

TRUST BOUNDARIES



User

Forms
headers
cookies
ajax
files

System

Files
services
Processes

Database

SQL
LDAP
web data stores

WHY XSS IS DIFFERENT

- Most injection flaws attack your server.

- XSS attacks the end user -- it runs arbitrary code in their browser.

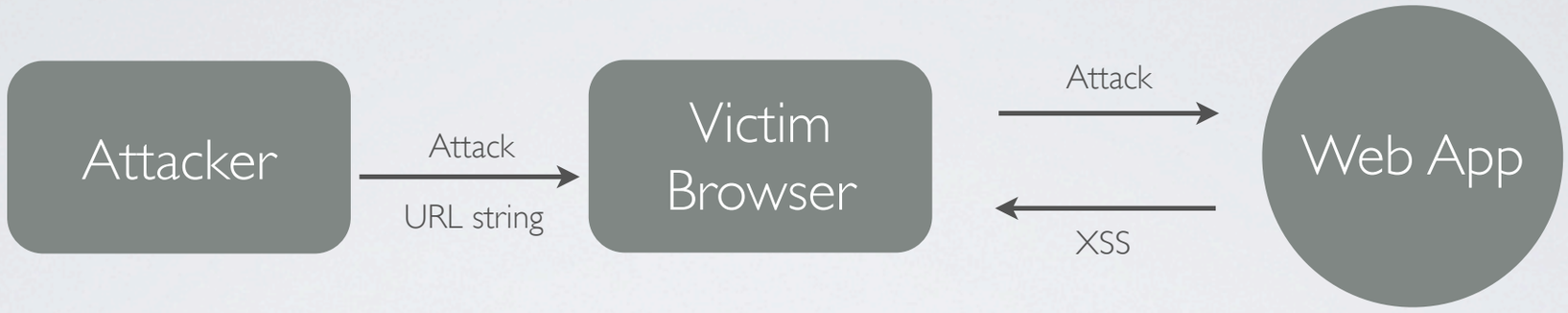
- The browser is behind your firewall and is acting within the user's security context.

YOU CAN DO SOME NASTY THINGS WITH JAVASCRIPT

- Javascript can control what appears on screen.
- Javascript has access to your history.
  - Sites often store session tokens in GET request.
- Javascript can intercept cookies.
- Javascript can enumerate your network.

XSS TAXONOMY

REFLECTED XSS

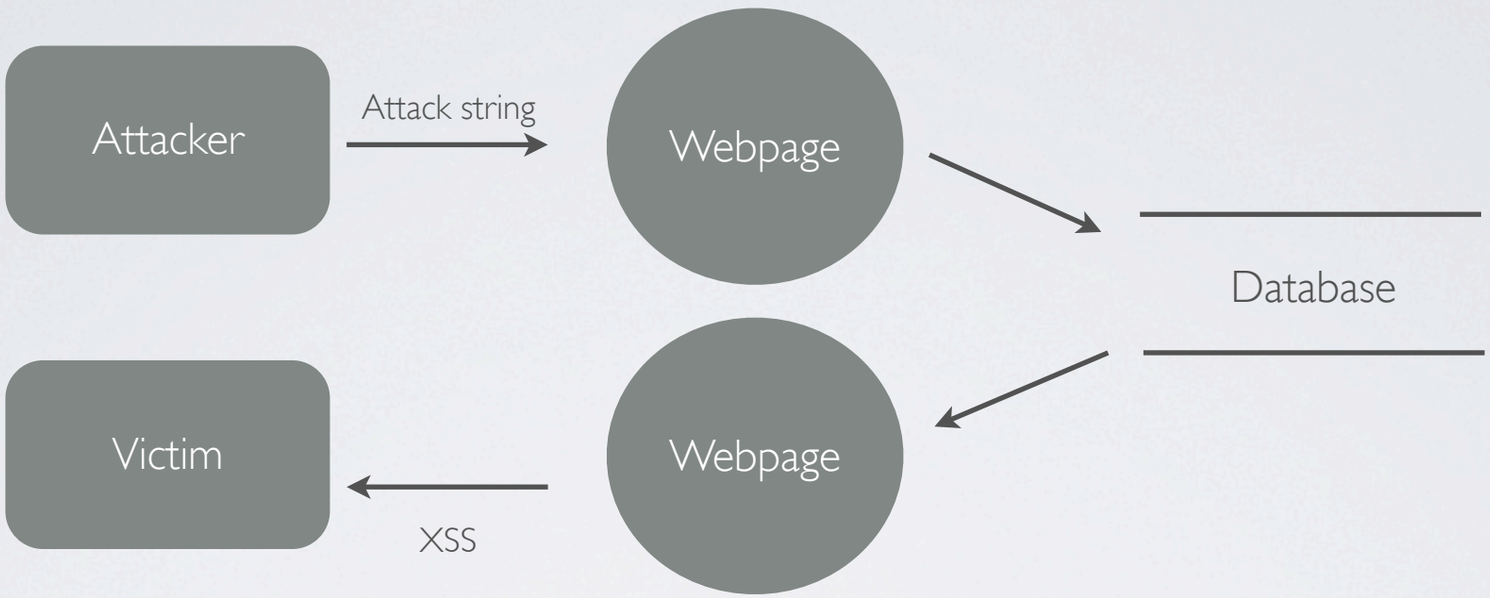


- Victim browser submits an "evil" request that contains javascript.
- The web application then reflects that javascript back to the victim browser, which then executes it.

REFLECTED XSS

- Attack does not persist across users, sessions, or pages.
  - It is only reflected back to the user that submits the malicious url.
- This is relatively easy to detect remotely via a scanner/fuzzer.

PERSISTENT XSS

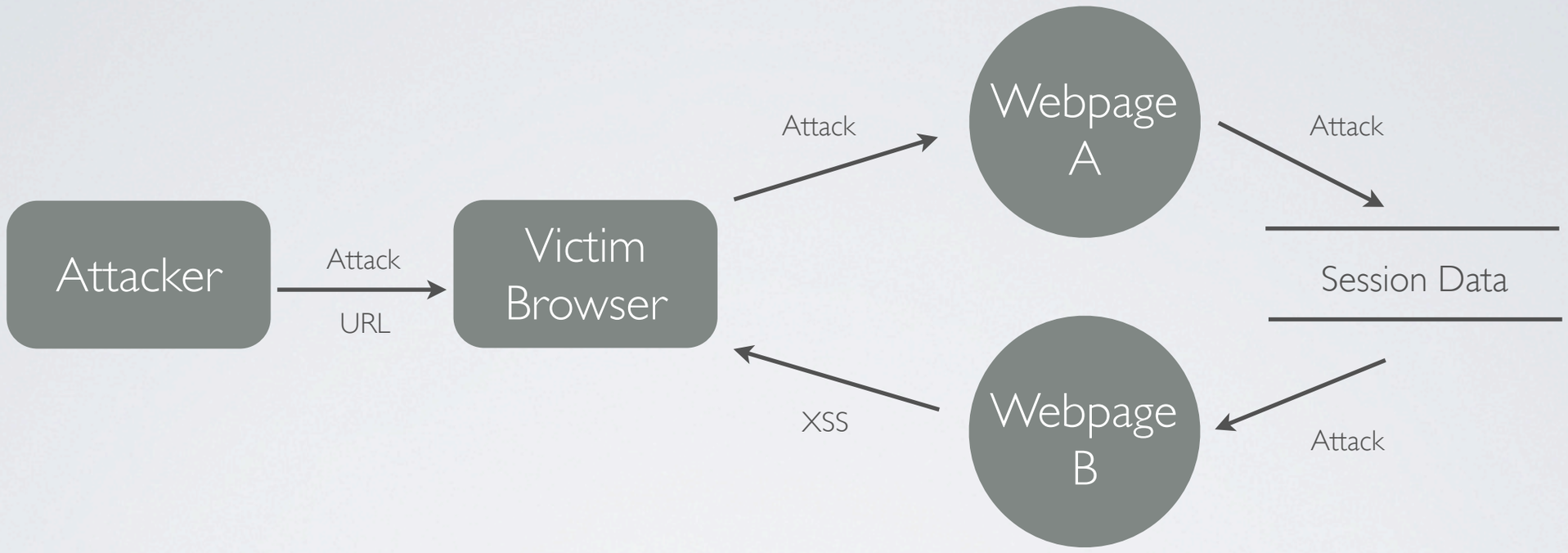


- An attacker stores an attack string in a web application.
- Visitors to that site access infected pages causing javascript execution.

PERSISTENT XSS

- Can persist across user, sessions, and pages.
  - It depends on application context and who can access the infected pages.
- Very difficult to detect remotely via scanning/fuzzing.
  - There are many paths through an application, scanning is too noisy for a production system.
- Best identified by code analysis.

HYBRID XSS



- Victim browser submits “evil” url.
- Attack string is stored as session data.
- Session data is returned to browser on another page, executing javascript.

HYBRID XSS

- Attacks are not persistent across sessions.
- Attacks do persist across pages.
- Attacks may persist across users.
  - Think about "who's online" functionality.
- Extremely difficult to detect remotely.
  - May require specific request sequences.

DETECTING INJECTION FLAWS REMOTELY

SCANNING FOR REFLECTED XSS VULNERABILITIES

- This requires a simple fuzzer.
  - Send an attack string.
  - Check for appropriately encoded returns.
- The most difficult part is maintaining a current attack string list.

SCANNING FOR PERSISTENT XSS VULNERABILITIES

- This sort of scanner is immature.
  - Requires mapping input cause vs. output effect without knowledge of application state.
- A persistent XSS scan is extremely noisy.
  - Each form has to have unique data submitted in order to map persistent data across the site.
  - Running an attack list against the mapped site creates one persistent record per attack string.

DETECTING INJECTION FLAWS LOCALLY

AVAILABLE TECHNIQUES

- Static code analysis
  - Examines the source code
- Dynamic analysis
  - Examines application state

STATIC CODE ANALYSIS

- Manual code review.
- Static code analyzers:
  - Pixy (PHP)
  - CAT.NET (ASP.NET)
  - CodeSecure (Java/ASP.NET/PHP)

WHAT STATIC ANALYSIS CAN DETECT

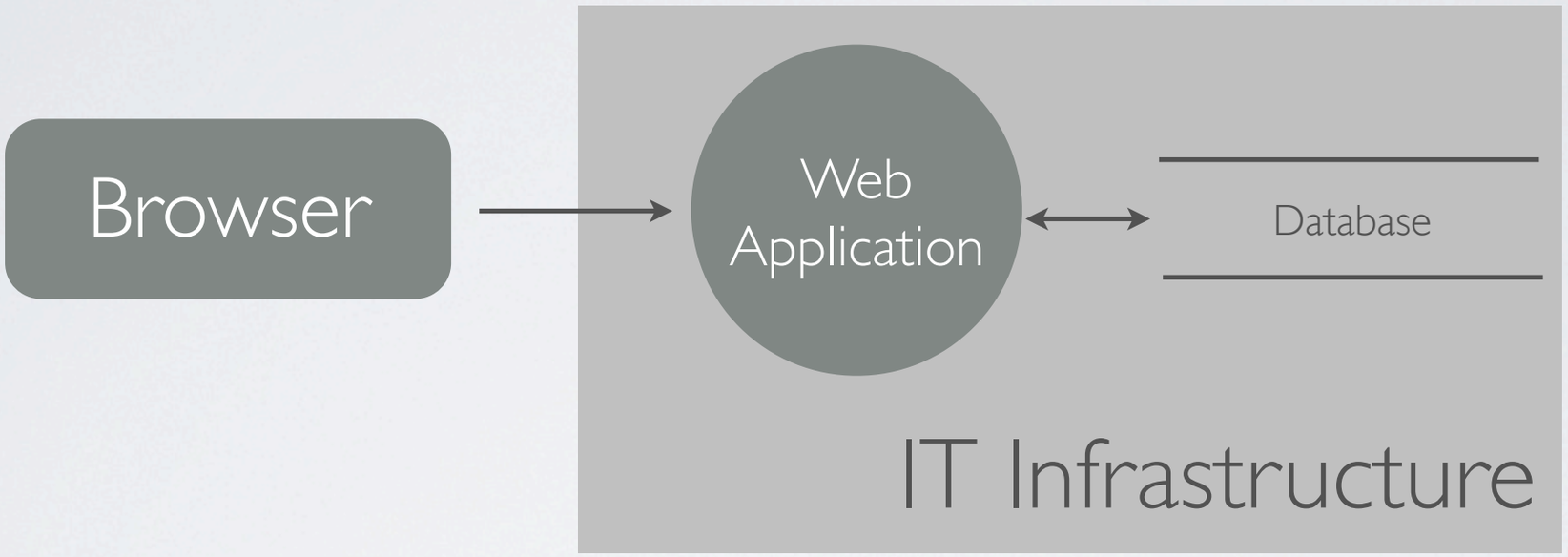
- Input sanitization
  - Really only boolean, doesn't evaluate quality.
- Output encoding
  - Are both application (html) and character encoding explicit?
- Data flow
  - Does anything skip sanitization or encoding?

DYNAMIC CODE ANALYSIS

- The only way to consider application state.
  - Remember that persistent and hybrid XSS are state-dependent.
  - Internal representation of data is tied to risk.
- There aren't a lot of existing tools.
  - Most existing are run-time IPS-style tools.

DEFENDING AGAINST XSS

THERE ARE THREE CONTROL POINTS FOR XSS



XSS can be mitigated at all three points.

THE BROWSER

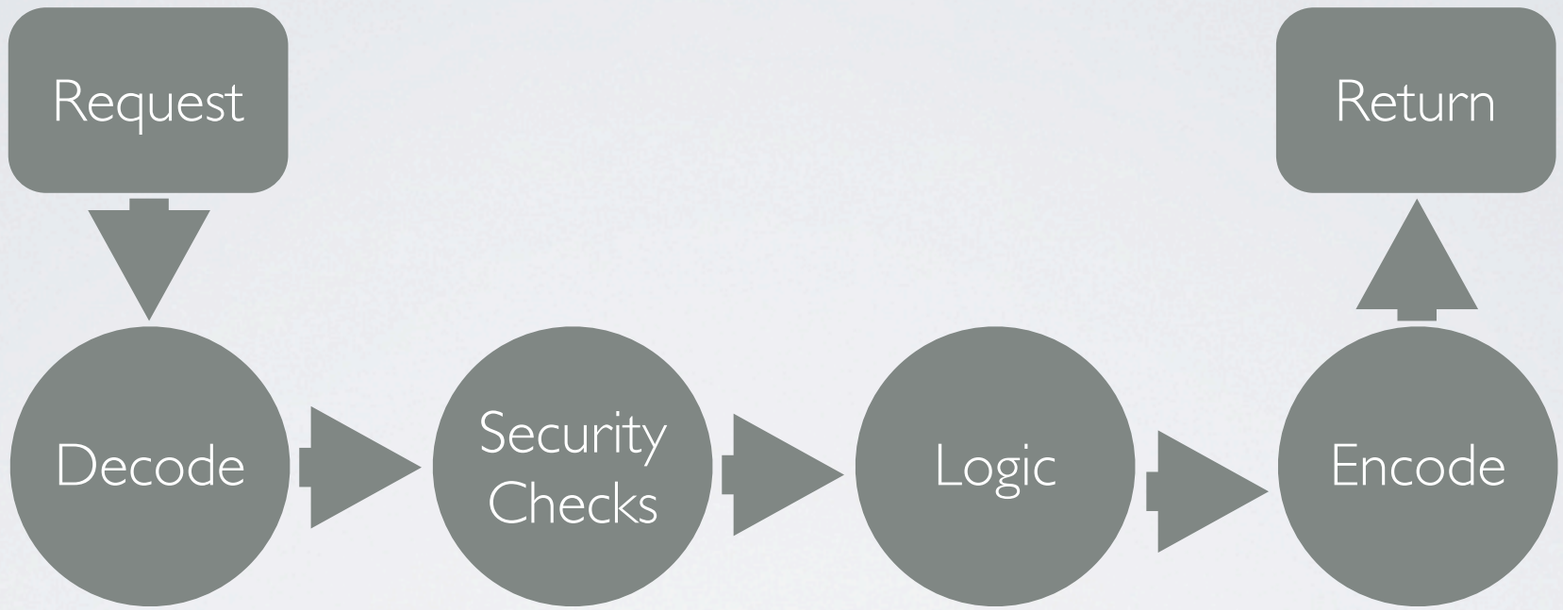
- Different browsers will execute different code snippets.
  - O9, IE6: <BODY BACKGROUND="javascript:alert('XSS');">
  - FF2, O9: <META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
- Current browsers are much better than previous generations.

IT INFRASTRUCTURE

- Web application firewalls.
- Intrusion prevention systems.
- Web application sandboxes.
- All of these are signature-based defenses.

WEB APPLICATION DEFENSE

IT'S ALL ABOUT PROCESS



- This requires a consistent and enforced architectural approach.
- JSP, PHP 3/4, and classic ASP don't lend themselves to this approach.

DECODE YOUR INPUT

```
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

- If the request specified an encoding, you might want to assume that encoding for parameters.
  - HTTP defaults to ISO-8859-1
- Convert the request to your application's internal representation.

SECURITY CHECKS

- This is where you apply your whitelists.
- Remember the principal of least privilege and use the most restrictive patterns possible.
- If you must accept markup tags, parse the tags and replace them with tokens.
  - Consider whitelisting any paths or URLs.

SANITIZING HTML

- There are a number of libraries available to do this for you.
  - Microsoft AntiXSS (.NET),
  - OWASP AntiSamy (Java, .NET, Python).
  - HTML Purifier (PHP)

WRITING YOUR OWN SANITIZATION LIBRARY

- Whitelist HTML tags.
  - Be as restrictive as possible.
  - If allowing high-risk tasks (such as <a> and <img>), consider whitelisting their targets.
- Replace all the allowed tags with symbols. Parse them for properties.
  - Store them this way.
  - Replace the symbols prior to encoding. This way, everything is constructed in your code and not passed from the user.

ENCODE YOUR OUTPUT

- Most commonly, encode using html entities.
- After entity encoding, replace your tokens with actual tags.
- If the final output is not html, encode accordingly.
  - XML if outputting AJAX, XAML, etc.

EXPLICITLY DECLARE YOUR CHARACTER SET

- Your web server can do it for you:
  - `AddCharset UTF-8 .php` (Apache .htaccess)
- If in doubt, override your web server:
  - `header('Content-type: text/html; charset=utf-8');` (PHP .htaccess)
  - `<%Response.charset="utf-8"%>` (ASP.NET)
  - `print "Content-Type: text/html; charset=utf-8\n\n";` (Perl)

DO YOUR SECURITY CHECKS ACTUALLY WORK?

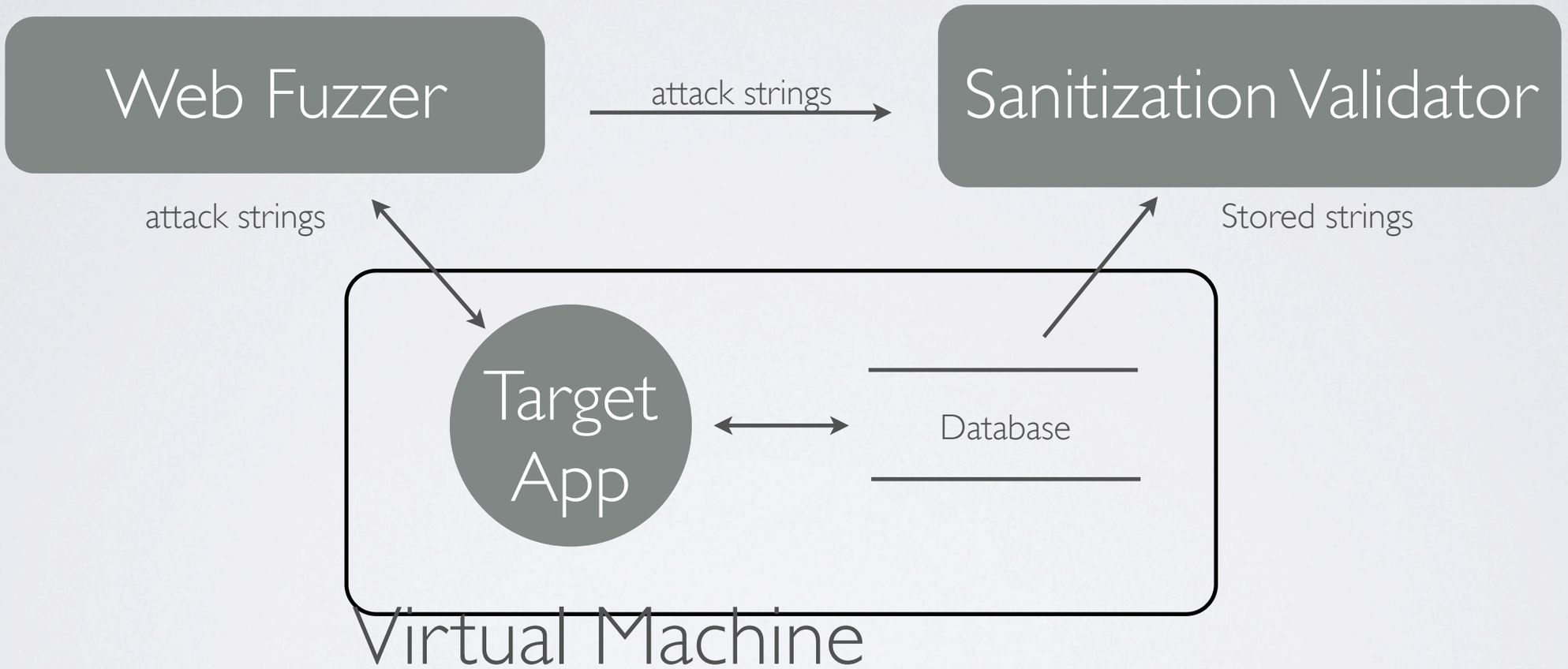
```
foreach ($_GET as $secvalue) {
    if ((eregi("<[^>]*script*\"?[^>]*>", $secvalue)) ||
     (eregi("<[^>]*object*\"?[^>]*>", $secvalue)) ||
     (eregi("<[^>]*iframe*\"?[^>]*>", $secvalue)) ||
     (eregi("<[^>]*applet*\"?[^>]*>", $secvalue)) ||
     (eregi("<[^>]*meta*\"?[^>]*>", $secvalue)) ||
     (eregi("<[^>]*style*\"?[^>]*>", $secvalue)) ||
     (eregi("<[^>]*form*\"?[^>]*>", $secvalue)) ||
     (eregi("\([^>]*\"?[^)]*\)", $secvalue)) ||
     (eregi("\"", $secvalue))) {
        die ("<center><img src=images/logo.gif><br><br><b>The html tags you attempted to use are not allowed</
b><br><br>[ <a href=\"javascript:history.go(-1)\"><b>Go Back</b></a> ]");
    }
}


foreach ($_POST as $secvalue) {
    if ((eregi("<[^>]*script*\"?[^>]*>", $secvalue)) ||(eregi("<[^>]*style*\"?[^>]*>", $secvalue))) {
        die ("<center><img src=images/logo.gif><br><br><b>The html tags you attempted to use are not allowed</
b><br><br>[ <a href=\"javascript:history.go(-1)\"><b>Go Back</b></a> ]");
    }
}
```

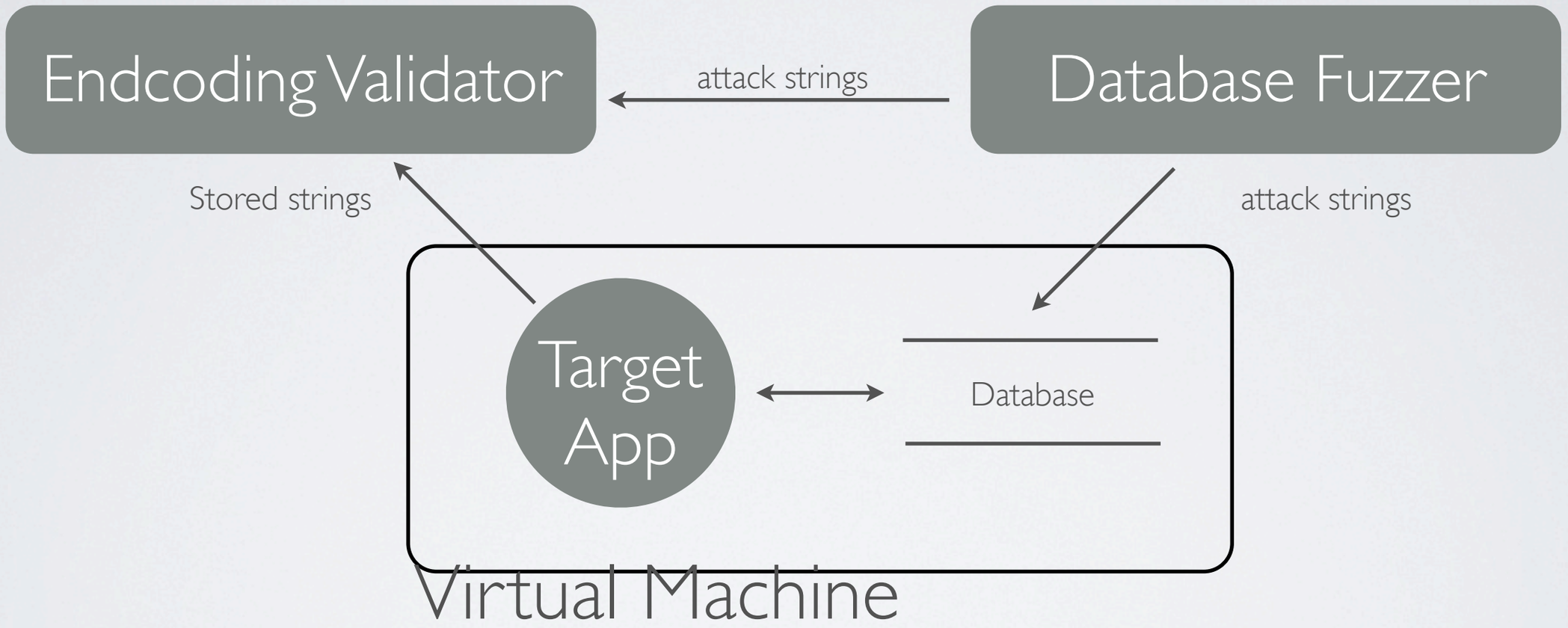
This is actual code in a popular CMS.

TESTING ARCHITECTURAL PATTERN IMPLEMENTATION

VALIDATES INPUT SANITIZATION



VALIDATES OUTPUT ENCODING



[DEMO]

TOOL BENEFITS

- Finds fundamental problems in software design
  - More complex attacks rely on either sanitization or encoding to be broken.
- Detects persistent XSS attack vectors.
  - In a dedicated VM, noise doesn't matter.

TOOL LIMITATIONS

- Only works with database-resident state data
  - Cookies, header, etc are also valid XSS threats.
- Only works against HTML forms
  - Can be extended to include AJAX, headers, cookies, etc.

TOOL PLANS

- Move it from PHP to C#.
- Create a PM-friendly interface so that anyone can run it.
- Add authentication integration:
  - Web forms, OpenID.
- Expand it to include non-form fuzzing.

WHY IS THIS HARD?

- Legacy applications and languages often don't lend themselves to this architectural pattern.
- Browser behavior is often unexpected and gibberish code can be executed.
- Application complexity makes it difficult to predict state, just like a client application.
- Applications and developers change over time.

CALL TO ACTION

- Write better code.
- Use current browser technology.
- Honestly assess your enterprise.
  - Do you have lots of legacy code?

I'M WEB 2.0

- Blog: http://hackerco.de
- LinkedIn: http://www.linkedin.com/pub/don-ankney/6/213/651
- Twitter: http://twitter.com/dankney
- E-mail: dankney@hackerco.de

BIBLIOGRAPHY

- Alshanetsky, Ilia. Php|Architect's Guide to Php Security|. Marco Tabini & Associates, Inc, 2005.

- Ernst, M., Artzi, S., Kiezun, A., Dolby, J., Tip, F., and Dig, D. "Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit State Model Checking." dspace.mit.edu (2009):

- Ernst, M., Kiezun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. "Automatic Creation of Sql Injection and Cross-Site Scripting Attacks." dspace.mit.edu (2008):

- Fogie, Seth, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. Xss Attacks: Cross Site Scripting Exploits and Defense. Syngress, 2007.

BIBLIOGRAPHY

- Johns, M., Engelmann, B., and Posegga, J. "Xssds: Server-Side Detection of Cross-Site Scripting Attacks." Proceedings of the 2008 Annual Computer Security Applications Conference (2008): 335-44.

- Stuttard, Dafydd, and Marcus Pinto. The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws. Wiley, 2007.

- Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. "Cross-Site Scripting Prevention With Dynamic Data Tainting and Static Analysis." Proceeding of the Network and Distributed System Security Symposium (NDSS'07) (2007):

- Wassermann, G., and Su, Z. "Static Detection of Cross-Site Scripting Vulnerabilities." Proceedings of the 30th international conference on Software engineering (2008): 171-80.

QUESTIONS?