

# **Intrusion detection for web applications**

# Intrusion detection for web applications

Łukasz Pilorz

Application Security Team, Allegro.pl

# Reasons for using IDS solutions

- known weaknesses and vulnerabilities
- balance between security and usability
- 3<sup>rd</sup>-party applications and libraries
- insecure client software
- additional layer of security
- fear, uncertainty, doubt

IDS, IPS or WAF?

# IDS purpose

- data source for post-intrusion analysis
- real-time intrusion investigation
- holy grail: intrusion prevention

How can we detect unknown attacks?

# Positive security model

- “accept known good” mantra
- allowed byte ranges
- regular expressions
- allowed variables whitelist

What about encoded (base64, weak encryption, multiple charsets) or complex (HTML, file upload) data?

# Positive security model

- when application changes, whitelist has to change too
- lots of alerts
- <http://p1.tld/p2/p3.php/p4/p5=p6,p7?p8&p9=p0>
- real-time protection? block them all!
- sanitizing wrong input could help

Why can't we do this in the application itself?

# It's easier to fix applications, than detect attacks

- usually true
- 3<sup>rd</sup> party software and libraries
- unknown attack methods
  
- security filters adding new vulnerabilities
- example: HTML filters

# HTML filters review – March 2008

Tested: 5 popular anti-XSS HTML filters (PHP)

Results:

- 3/5 vulnerable to XSS (+1 already known 0-day)
- 2/5 included PHP code execution bugs (kses, htmLawed)
- alternative syntax like Textile or Markdown also not safe from XSS



# Negative security model

- blacklist detection rules
- far less alerts
- classification by attack type, priority, etc.
- generic rules: often too general, false positives
- specific rules: very limited, often outdated

How to detect unknown attacks?

# Examples

- Snort – known exploits
- ModSecurity Core Rules – generic
- PHPIDS – generic, focused on XSS

# PHPIDS

- LGPL licensed IDS library for PHP applications
- impact rating for each malicious request
- could be added in `auto_prepend_file`, without modifying application code
- attempts to detect unknown attack patterns

<http://php-ids.org/>

# **IDS vs OWASP Top Ten**

# What are we trying to detect?

- automated exploits
- automated vulnerability scanners
- manual attacks
- uncommon user behaviour
  
- intrusion vs vulnerability testing

How to recognize source type?

# A1 - Cross Site Scripting (XSS)

- most common: `<script`, `document.cookie`
- dangerous HTML tags and attributes
- breaking out of HTML attribute
- JavaScript keywords
  
- comparing request and response
  
- PHPIDS regular expressions

# XSS from attacker's view

- needs just one byte to detect vulnerability, e.g. “, <, (
- easy to make it look innocent

```
<a href="http://tested.site.tld/page.php?id=article&quot;>Interesting article</a>  
<script>[Google Analytics]...
```

- usually needs at least several requests to prepare working attack (for custom application)

# XSS – detection

- hard to detect less common vulnerability testing patterns
- recognizing malicious XSS code is easier
- time window between finding vulnerability and developing exploit
- real-time detection could prevent attack

How to detect DOM-based XSS or 3<sup>rd</sup> party JavaScript/CSS modifications?



# XSS – reducing false positives

- different rules for public and private application sections
- check for persistent XSS after HTML filtering (response buffering or PHPIDS)
- don't alert when only single keyword/char matches rule (skip non-malicious XSS)
- raise impact rating for suspicious or missing Referer headers
- don't even think about “trusted IPs”

# A2 – Injection Flaws

- paranoid mode: blocking semicolon and quotes
- checking for SQL (or other language) keywords
- 2.0, 2-0, 2-1, 2:[query]
- val'||', val';[query]
- 2 AND 1=1, 2 AND 1=2
- 2 UNION...SELECT [query]
- /\* ... \*/, /\*!....\*/
- “page.tld/page?var=1/\*&UNION SELECT\*/”

# SQL Injection

- relatively easy to detect malicious attacks
- many false positives, if we want to detect vulnerability testing
- good results with whitelisting
  
- reducing false positives by checking traffic between application and database (or in the application, before executing query)
- real-time reporting of SQL query errors

# Command/Code Injection

- much wider range of malicious code than for SQL Injection
- detect vulnerability testing, not exploits
- reducing false positives by eliminating known vectors
  
- common commands and functions
- ` , {\${ , <? , <%
- real-time reporting of application errors

# Other injection flaws

- LDAP
- XPath
- XSLT
- HTML
- HTTP

# A3 – Malicious File Execution

- affects mostly PHP
- external URL in request (http://, ftp://)
- wrappers (data:, php:, ogg:, zlib:, zip:)
- /var/log/httpd/
- /proc/self/environ + User-Agent
- /, ../
  
- upload containing PHP code
- upload filename & extension

# A4 – Insecure

## Direct Object Reference

- easier to fix than detect
- whitelisting often doesn't help
- we can try to detect data harvesting tools
- multiple requests to the same page, with different set of parameters
- repeating requests to a single page or a small subset of pages
- small mistakes in automatically generated requests (Referer, null bytes, missing headers or cookies)

# A5 – Cross Site Request Forgery (CSRF)

- why black hats love CSRF?
- again, it's really easier to fix than detect
  
- external or missing Referer header
- missing cookies
- Accept header
- user trying to perform action while logged out
- user trying to remind password while logged in
- broken application flow
  
- Referer-less redirects, clickjacking



# A6 – Information Leakage and Improper Error Handling

- monitoring outbound traffic (e.g. ModSecurity)
- application code, HTML comments, error messages (esp. SQL)
- 3<sup>rd</sup> party software may leak undocumented or non-standard error messages
- what information should be treated as leakage (and how IDS knows it)?
- Blind SQL Injection

# Forcing errors

- `var[]=1`
- `1.1, 1x, ./1, /1`
- `“, ’, !, %0A, %00`
  
- wrong type of data
- wrong format of session identifier
- DoS
- ...
  
- too many possibilities to check requests

# A7 – Broken Authentication and Session Management

Session hijacking detection

- another one that is easier to fix in the application itself (or rather “fix”)

After identifier is stolen:

- IP address change during session
- headers changed/missing during session

Before:

- tampering session identifiers
- XSS

# Session hijacking – attacker's view

- sniffing traffic  
Spoof IP, everything else you already have.
- XSS  
You don't need to hijack session identifier, just force the victim to do whatever you wanted.
- Referer header

# A8/A9 – Insecure Cryptographic Storage and Communications

- not much to do for an IDS (at least on the server side)
- passing base64-encoded or weakly encrypted values to the client
- WAF protection against tampering
- may be decrypted on client side and leak information
- general brute-force attacks detection

# A10 – Failure to Restrict URL Access

- IDS has no information about user rights in the application
- known vulnerabilities in libraries/include files
- brute-force detection may deal with fuzzing
- broken application flow
- whitelisting
- IPS/WAF as a hotfix solution

■ ■ ■

# Log, block, alert

## 3-tier solution

### Tier 1:

- log everything you can

### Tier 2:

- detailed log of detected attack attempts

### Tier 3:

- possible intrusions and bypasses



# Tier 1

- log everything you can
- all application errors  
(with their context)
- full requests  
(URL, headers, cookies, body)
- full responses  
(HTTP code, headers, body)

# Tier 2

- detailed log of detected attack attempts
- IDS alerts
- combined data from several sources
- including vulnerability testing patterns
- including blocked/sanitized requests
- optionally: requests following blocked one

# Tier 3

- possible intrusions and bypasses
- alerts that require manual verification
- generate as much as you are able to check manually
- skip blocked requests