Security II - More Server-Side Security

Stefano Calzavara

Università Ca’ Foscari Venezia

March 26, 2020
Introduction

In this lecture, we will focus on three classes of problems which affect the server-side logic of the web application:

1. **Server-Side Request Forgery**: abuse the web server as a confused deputy to make it take actions under the attacker’s control

2. **XML External Entities**: abuse some dangerous features of the XML file format to trigger server-side actions under the attacker’s control

3. **HTTP Parameter Pollution**: confuse the web application on HTTP parameter parsing to force unintended behavior

We will not discuss database security, since it was already covered in the first module (Security I).
Server-side request forgery (SSRF) is a web security vulnerability that allows an attacker to induce the server-side application to make HTTP requests to an arbitrary host of the attacker’s choosing.

- This is dangerous, because it makes the server a *confused deputy* and enables *privilege escalation* attacks.
- Typical targets: the local host or other back-end servers sitting on the same local network, protected by a firewall.
- We will discuss soon other threats coming from SSRF.
Why do we need server-side requests in web applications?

- **Preview of resources**: try sending a link over Slack
- **Caching / proxies**: to preserve privacy of the end users
- **Data import**: just search for something on Google Images

... and possibly more use cases!
SSRF: Attacking the Local Host

Let’s assume a front-end server at www.foo.com gets stock information requests and forwards them to a back-end server, which then provides the result. The front-end accepts POST requests with parameter:

```
stockApi=http://stock.foo.com/prodId%3D6%26storeId%3D12
```

The attacker can forge a request with parameter:

```
stockApi=http://127.0.0.1/admin
```

Since the administration interface of the web app is locally accessible, the attacker performs privilege escalation through the response!
A variant of the same attack can be used to target other machines sitting on the same local network:

```
stockApi=http://192.168.0.68/admin
```

These machines are not visible from the Internet, but can be accessed by the confused server who shares their local network.

This form of attack is getting a lot of traction in the recent years thanks to the rise of IoT devices...
SSRF: Other Variants

SSRF can also be abused to:

- **Attack remote servers**: the confused server is fooled into sending malicious requests to other remote servers, so that the attack is not coming from the attacker’s machine

- **Bypass SOP**: the confused server is fooled into fetching malicious content from the attacker’s server, so that the attacker gets script capabilities in the server’s origin

A dangerous vulnerability, which should be readily fixed!
Preventing SSRF: Black-Listing

Some applications block input containing hostnames like 127.0.0.1 and localhost, or sensitive paths like /admin

- the localhost ranges from 127.0.0.0 to 127.255.255.255
- alternative IP representations exists, for example 127.1
- the attacker can register a domain and make it resolve to 127.0.0.1
- different encodings of URLs, e.g., double-encoding attacks
- case variations and other quirks in URL parsing libraries [1]
Parsing URLs is Hard!

The full syntax of URLs is surprisingly complicated...

http://user:pass@192.168.0.1:80/path?foo=one&bar=two#frag

How shall we parse http://google.com#@evil.com?

- Option 1: request to evil.com with user google.com#
- Option 2: request to google.com with fragment @evil.com

Just one example, see [1] for other nasty details!
Preventing SSRF: White-Listing

Some applications only allow input that matches, begins with, or contains, a whitelist of permitted values

- fake credentials: https://good.com@evil.com
- fragment identifier: https://evil.com#good.com
- subdomains: https://good.com.evil.com
- again, quirks in URL parsing libraries
SSRF and Open Redirects

An open redirect vulnerability happens when a web application redirects users to an attacker-controlled URL. Normally low severity, but quite dangerous in the context of SSRF.

```
stockApi=http://stock.foo.com?path=127.0.0.1/admin
```

This attack bypasses filtering because:

- the front-end sends a request to http://stock.foo.com, which is allowed by the filter
- the back-end redirects the front-end to 127.0.0.1
- the response of the redirect is returned to the attacker
XML External Entities

XML external entities is a web security vulnerability arising from the abuse of little known, dangerous features of the XML format.

```
<?xml version="1.0"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Buy milk</body>
</note>
```

XML is a markup language, which makes use of tags much like HTML. The validity of an XML file is determined by its Document Type Definition (DTD).
Example: DTD

```xml
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)> 
  <!ELEMENT to (#PCDATA)> 
  <!ELEMENT from (#PCDATA)> 
  <!ELEMENT heading (#PCDATA)> 
  <!ELEMENT body (#PCDATA)> ]>

<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Buy milk</body>
</note>
```
XML Entities

In XML, an entity is just a binding between a name and a value, defined in the DTD.

```
<!ENTITY wife "Jani">
```

The name of an entity can be mapped to its corresponding value in the XML document by using a special syntax, e.g., `&wife;`

An external entity binds a name to a URI.

```
<!ENTITY server SYSTEM "http://stock.foo.com">
```
XML Billion Laughs Attack

Thanks to colleague Ben Stock from CISPA for sharing this...

<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">  
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">  
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">  
<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">  
<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">  
<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">  
<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">  
<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">  
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">  
]]>
<lolz>&lol9;</lolz>
XXE: Retrieving Files

Suppose a shopping application checks for the stock information of a product by submitting the following XML to the server:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

Stealing the password file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]> 
<stockCheck><productId>&xxe;</productId></stockCheck>
```
XXE: Performing SSRF

Since an XML external entity can point to any URI, e.g., resolving to a local IP address, it is possible to abuse XXE to perform SSRF.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "http://192.168.0.68/admin">
]> 
<stockCheck><productId>&xxe;</productId></stockCheck>
```
Do not underestimate the amount of XML information which is still exchanged nowadays!

Many common file formats are based on XML:

- Document formats like DOCX
- Image formats like SVG

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg height="60" width="200">
  <text x="0" y="15" fill="red" transform="rotate(30 20,40)">
    I totally hate SVG!
  </text>
</svg>
```
If you know the enemy and know yourself, you need not fear the result of a hundred battles (Sun Tzu, The Art of War)

- now that you know about the existence of XXE, check the details of your XML parser!
- many modern XML parsing libraries disable support for XXE, unless you explicitly relax this security restriction
- yet, libraries might be vulnerable to the billion laugh attack!
- disallow DTD definitions in XML files and use a static, local DTD
HTTP Parameter Pollution

HTTP parameter pollution (HPP) is a vulnerability enabled by the HTTP parameter parsing APIs of web programming languages.

```php
x = $_GET[‘user’]
```

What happens if the GET parameter contains two user parameters?

- the first occurrence?
- the last occurrence?
- a list of occurrences?

The truth is... this varies a lot!
Parameter Parsing in Web Frameworks

The Web is not a place for the weak-hearted...

<table>
<thead>
<tr>
<th>Framework</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP</td>
<td>All occurrences (,)</td>
<td>$v_1, v_2$</td>
</tr>
<tr>
<td>JSP</td>
<td>First occurrence</td>
<td>$v_1$</td>
</tr>
<tr>
<td>perl</td>
<td>First occurrence</td>
<td>$v_1$</td>
</tr>
<tr>
<td>PHP</td>
<td>Last occurrence</td>
<td>$v_2$</td>
</tr>
<tr>
<td>Python</td>
<td>List of occurrences</td>
<td>$[v_1, v_2]$</td>
</tr>
</tbody>
</table>

Web server and application may differ in understanding of parameters!
Assume the university server hosts a PHP application which processes POST requests, including parameters of the following form:

```
examId=12&finalMark=25&studentId=123456
```

How can you always pass the exams with the highest mark? Assume you can manually input your `studentId` (443256) during signup.
Assume the university server hosts a PHP application which processes POST requests, including parameters of the following form:

```
examId=12&finalMark=25&studentId=123456
```

How can you always pass the exams with the highest mark? Assume you can manually input your `studentId` (443256) during signup.

Enjoy your free exams by signing up as: `443256&finalMark=30`. 
HPP: Example

Infamous example at Blogger...

```
POST /add-authors HTTP/1.1

security_token=attackertoken&
blogID=attackerblogidvalue&
blogID=victimblogidvalue&
authorsList=attacker%40gmail.com&
ok=Invite
```

Permission check on the first occurrence of blogID, but target blog extracted from the second occurrence of blogID.
Defending Against HPP

HPP is relatively easy to defend against... once you know it exists!

- check the documentation of your web development framework
- if the API gives you back a list of parameters, you have all the information you need
- otherwise, parse the parameters manually and check that none occurs multiple times
- encode the & characters to avoid the discussed attack
Cookie Shadowing

A variant of HPP in the context of cookies is called cookie shadowing [2]

- cookies have a scope, depending on the combination of the Domain, Secure and Path attributes
- it is possible for two cookies to have the same name, but different scope, e.g., a host-only cookie and a domain cookie both called sid, which are both received by the server
- the attacker can exploit this to force the server into preferring an attacker-created cookie over a legitimate cookie with the same name
- subtle problem: this depends on both the client and the server!
Example: Cookie Shadowing

Credit card stealing against China UnionPay [2]

- single session cookie `uc_s_key`
- possibility to associate a credit card number to an existing account to simplify future payment processes
- the attacker can shadow the victim’s cookie with his own cookie to make China UnionPay get the association wrong, i.e., associate the victim’s credit card to the attacker’s account
- no visual indicator of the account identity at the association interface
References

Orange Tsai.
A new era of ssrf - exploiting url parser in trending programming languages!
Black Hat 2017.

Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver.
Cookies lack integrity: Real-world implications.