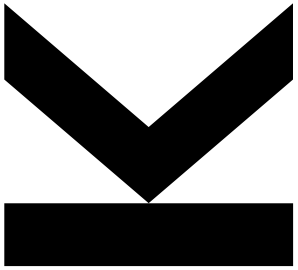
X-type (cross-...) attacks



JOHANNES KEPLER
UNIVERSITÄT LINZ

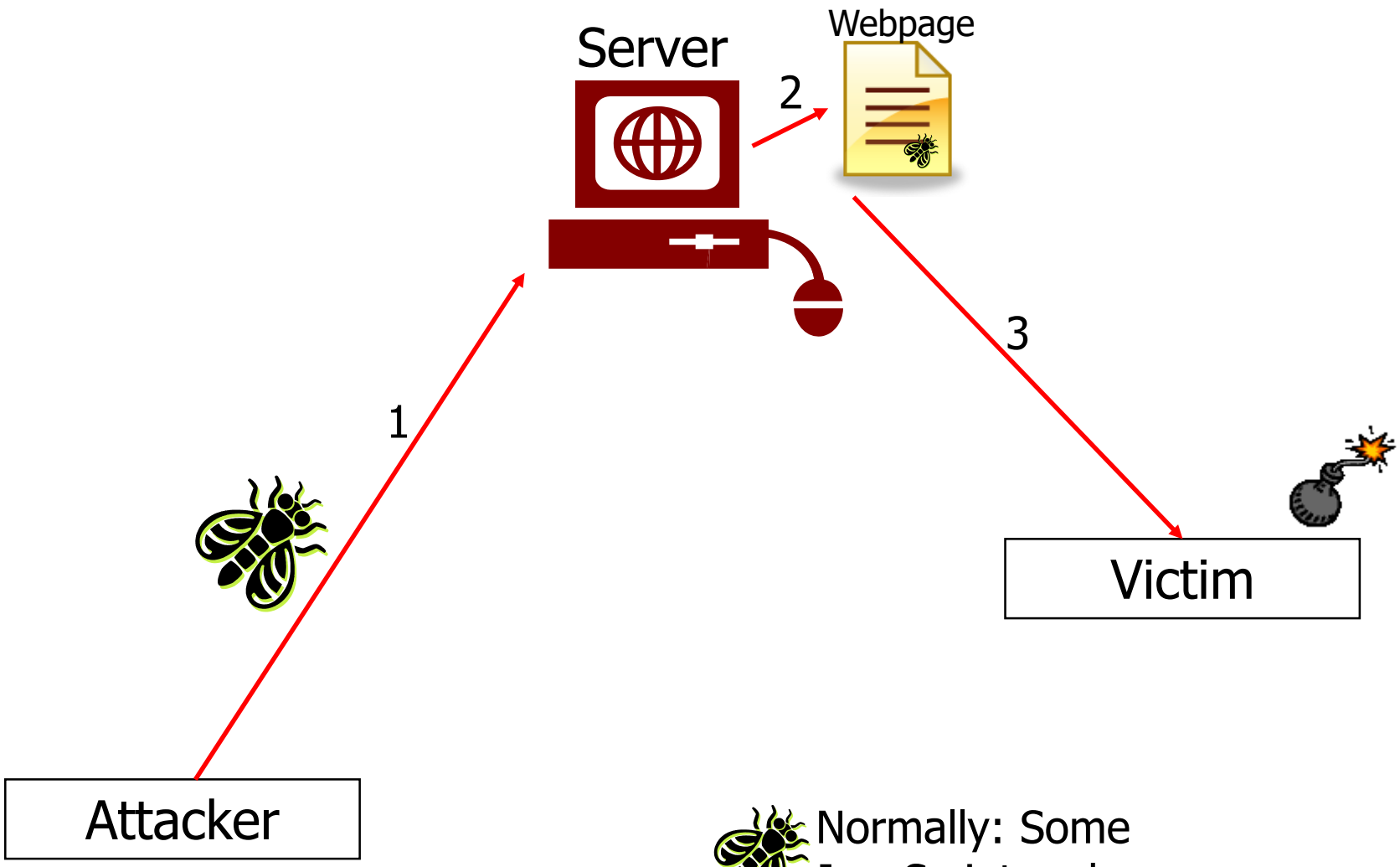
Michael Sonntag
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

Cross-Site Scripting (XSS)

JKU

Cross-site-scripting (XSS)



Cross-site-scripting (XSS)

- Code injection by malicious users into someone else's (=third party) web application, to be viewed/executed by end users (=victims)
  - Typically a problem of bad input validation!
- XSS example:
  - Online banking site with discussion forum
  - Post a message with JavaScript code embedded in it
  - Every user viewing this message will execute this code in her own browser; within the context of the banking site (=SOP!)
- Note: The URL is perfectly fine!
  - **Browser security features will not help here!**
  - **Bypasses access controls and same-origin-policy!**
  - **Encryption (TLS) and certificates will not help at all!**
- 2007: Approx. 80% of all security vulnerabilities were XSS
  - Other sources: 90% of all websites contain one of these
  - 2018: 25% of all detected vulnerabilities (company: Netsparker)

“Stored" (1) or "Reflected" (2) XSS

- Stored: “Store” the script on the site
  - Data entered by the user is stored in a DB and sent “back“ whenever a certain page/article/… is accessed
    - I.e. the stored data is used to construct the response
  - Huge multiplication factor: 1 site → thousands of users!
- Reflected: Injecting a script which is “bounced” back
  - Could be reflected by a search result page, some quote, or an error message
    - Any response which contains at least some part of the user input
  - Can be encoded in the URL
    - So it might be provided from site-externally!
    - Simple to exploit: Just bring someone to click on this special link
    - Note: This code can be encoded in the URL, e.g. by obfuscation or link shorteners, to be unrecognizable as program code!
    - Example: Links in Spam messages

DOM-based (3) XSS

- Injected code is executed through modifying the DOM in the victims browser used by the original script
  - Normal script + “strange” input data = unexpected results
- The page itself is exactly as it should be, but the DOM model created in the client is different than it should be
  - Servers can detect some kinds (below: in request URL)
- Example: Code to select language
  - Select your language: <select><script> document.write("<OPTION value=1>"+document.location.href.substring(document.location.href.indexOf("default=")+8)+ "</OPTION>"); document.write("<OPTION value=2>English</OPTION>"); </script></select>
  - Normal URL: **http://www.some.site/page.html?default=French**
  - DOM-based XSS attack: Get the user to click on the following URL **http://www.some.site/page.html?default=<script>alert(document.cookie)</script>**
  - The following URL is requested (=document.location in result): http://www.site.com/page.html?default=<script>alert(document.cookie)</script>
  - When rendering the page, “alert(document.cookie)” is executed!
  - Note: The **page content** sent over the network does not contain the code “alert(document.cookie)” at all, as this is retrieved dynamically from the address bar (=URL) only in the user’s browser! But it **IS** sent over the network → request string
- Especially vulnerable: document.location, anchors (URL after “#”)

XSS examples: Filter evasion

- Tries several attacks at once (different quotes etc.)
  - ';alert(String.fromCharCode(88,83,83))//';alert(String.fromCharCode(88,83,83))//"; alert(String.fromCharCode(88,83,83))//";alert(String.fromCharCode(88,83,83))//--></SCRIPT>">'><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
- Short version: ' ';!--"<XSS>=&{()}
- Image; no quotes/semicolon: <IMG SRC=javascript:alert('XSS')>
- Obfuscation with Grave accent: <IMG SRC=`javascript:alert("XSS")`>
- Malformed IMG tag: <IMG " " ">"><SCRIPT>alert("XSS")</SCRIPT>">
- No quotes: <IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
- Style tag: <STYLE>@im\port'\ja\vasc\ript:alert("XSS")';</STYLE>
- Null character (needs to be injected specially):
  - perl -e 'print "<IMG SRC=java\0script:alert(\"XSS\")>";' > out
- Find needed characters in existing page source, extract and insert by JavaScript

Source: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Cross-site-scripting: Consequences

- What is the result? XSS can do the following:
  - It can steal cookies and session tokens
  - It can present a login-form
    - With the information entered being sent to the attacker!
  - It can read and change all data on this page
  - It can be used as a proxy, for DoS, or port mapping attacks on the local network or third-party sites
  - All actions are performed as if the code came from a trusted site
- Encoding possibilities to hide the code:
  - Using Unicode, entities, escaping…
  - Can avoid using "<" or ">"
  - ActiveX, Flash and similar techniques may also be used
- MySpace XSS worm: 1 million victims in <24 hours!
  - Stored XSS; viewing an infected profile was sufficient

XSS Example: MySpace worm (excerpt)

```
var B=String.fromCharCode(34); ← Double quotation mark “
var A=String.fromCharCode(39); ← Single quotation mark ´
function g() { … retrieve complete code of page and return as string … }
var AA=g();
var AB=AA.indexOf('m'+'ycode'); var AC=AA.substring(AB,AB+4096);
var AD=AC.indexOf('D'+'IV'); var AE=AC.substring(0,AD);
```

- Extract code of worm from the whole page into variable AE

```
if(AE) {
  AE=AE.replace('jav'+'a',A+'jav'+'a’);
  AE=AE.replace('exp'+'r)','exp'+'r)'+A);
```

- Prevent detection: Split „dangerous code“ into separate strings
- MySpace removed the string „javascript“, quotes… from any input
  - Plus a few other strings (<script>, <body>, onClick, “, ´, \“, \´,…)

```
  AF=' but most of all, samy is my hero. <d'+'iv id='+AE+'D'+'IV>,
```

- This is the text which is inserted into the page (=the “malware”)!

```
}
```

The “worm” part → Malware contains itself

http://www.bindshell.net/papers/xssv/myspace/myspaceviruscode.txt

XSS Example: MySpace worm (excerpt)

```
…
AG+=AF;
```

- AF is the string including the worm code!

```
var AR=getFromURL(AU,'Mytoken');
var AS=new Array();
AS['interestLabel']='heroes';
AS['submit']='Submit';
AS['interest']=AG;
AS['hash']=getHiddenParameter(AU,'hash');
```

- MySpace generated a random hash on a GET page, which had to be passed via the POST to actually add a friend
- Get this page first (not shown here) and extract the token into AR

```
httpSend('/index.cfm?fuseaction=profile.previewInterests&Mytoken='+AR,
    postHero,'POST',paramsToString(AS))
```

- Confirming the addition is not shown here, but works similarly!

http://www.bindshell.net/papers/xssv/myspace/myspaceviruscode.txt

XSS Example: MySpace worm (excerpt)

- The resulting page did look like this:

  <div id=mycode style="BACKGROUND: url('java
  script:eval(document.all.mycode.expr)')„
  expr="var B= … ← See previous slide!

  …

  return true}"></DIV>

- Very important: Line break between “java” and “script”!
  - This enabled the code to **not be filtered** out, but **still be executed** within the browser!
- Script is stored in “expr” so single quotes can be used in it
  - Otherwise both single and double quotes would already have been used and we could use neither!
  - In “expr” only double quotes have been “used up”

http://www.bindshell.net/papers/xssv/myspace/myspaceviruscode.txt

Cross-site-scripting: Prevention

- Never try to filter out offending content, it just won't work!
- Always escape everything you write to the user
  - Escaping <, >, (,), #, &, ", ‘, /, \ significantly **increases** security!
    - Result: **No** HTML can be embedded at all!
    - Use Wiki technologies (“[…]” → link) → Customs "tags" which are converted to explicit and known HTML tags on output
      - But have no HTML meaning themselves → Slip through = harmless!
    - Note: Entity encoding **alone** is often **not** enough!
      - Example: Inserting input into <script> tags, event handlers, CSS…
  - "Tainting" may help → Automatic tracking of "external" data
- Always validate all user input
  - Whitelist: Only accept data exactly matching expect. format
- Cookies: Tie to IP address and mark as "HttpOnly“
- Users: Enter URLs manually/through bookmark
  - Don't click on links in spam messages/message boards
  - Turn off JavaScript and disable plugins

Cross-site-scripting: Prevention

- Complete prevention is very complex!
  - SQL injection is trivial to protect against in comparison!
- Problem: HTML is very wide and allows all kinds of “hacks”
  - Background: It’s complex; browsers are very fault-tolerant
- Best and perfect solution:
  - Whatever users submit, it’s never sent to a client
    - This advice, while correct, is probably not very useful…
- So what to do?
  - Escape all user-submitted content before sending it out
  - This is not easy: Depending on the location of the content in the HTML file the escaping required differs
- Some things cannot be protected against
  - You have to live without them (**only if** they **include user input**)!
    - Example: eval, execScript, setTimeout, setInterval functions
    - They produce code from strings (your code = OK; user data = Never)!

Examples of incorrect HTML

- The following examples are incorrect HTML, but will still “work” in a browser, creating unintended consequences and allowing XSS
  - var data=‘</script><script>alert(1);//’;
    - Removing the comment leads to a “valid” string → no XSS
      - “Literal not terminated before end of script” in both variants
  - var string=‘<img src=b onerror=alert(2);>’;
    document.getElementById(“target”).innerHTML=string;
    - Note that the first line here is still a correct string; only when putting it into the document the attack becomes active!
  - var string='\u003Cimg src=1 onerror=alert(3)\u003E';
    document.getElementById("target").innerHTML=string;
    - We don’t need any special characters like “<“; we can encode them as UTF characters (\u????)!
  - var string='\x3Cimg src=1 onerror=alert(4)\x3E';
    document.getElementById("target").innerHTML=string;
    - Almost the same as above, but directly encoded as hexadecimal…

See https://en.wikibooks.org/wiki/Web_Application_Security_Guide/Cross-site_scripting_(XSS)

Examples of incorrect HTML

- More examples:
  - <script>function callFunc(x){}</script>
    <img src="some.gif"
    onerror="callFunc('');alert(5);callFunc('');" />
    - Note the incorrect function call: Starts with a single quote, but is not terminated until the end of the second function call
    - Also, the picture does not exist and therefore this is executed
    - If you put code into the function you notice that it is called twice: before and after the XSS execution
  - <script>
    /*
    //var data='</script><script>alert(6)<!--';
    - Even inserting text into a comment is not safe…
  - var string='<img src='1' onerror=alert(7);>';
    document.getElementById("target").innerHTML=string;
    - Still inside a comment (according to the editor ☺)

Still executed despite comments! But „comment“ of ending ‘ remains necessary. (as above)

See https://en.wikibooks.org/wiki/Web_Application_Security_Guide/Cross-site_scripting_(XSS)

Cross-site-scripting: Prevention

- Externalize all scripts and use CSP
  - Try to avoid whitelisting by nonce
    - Can be exploited in some cases (see example)
- Several rules by OWASP (and me):
- -1: Never insert JS code from another site into your page
  - No matter how you obtain it, as a URL parameter, request response, TCP connection… unless you **fully trust** it
  - Reason: You do not control the content, it can change at any time
    - jQuery etc from central servers (=caching!) → Do you trust them?
    - Referer header: They get nice information on your site traffic (who, how many, countries etc)
    - Encryption to ensure no modifications happen during transport
    - Always newest version → Still compatible?
    - Fixed version → Still present?
    - Optimized version equivalent to human-readable one?

Cross-site-scripting: Prevention

- 0: Never insert untrusted data except in allowed locations
  - Directly in a script <script> ... **UNTRUSTED** … </script>
    - Text inside is something different; here we are talking about code
  - Inside HTML comments <!-- … **UNTRUSTED** … -->
  - In attribute names <div na**UNTRUSTED**me="…">
  - In tag names <di**UNTRUSTED**v id= …>
  - Reason: There are too many ways to get out of these, so guaranteed escaping is practically impossible
  - You should not need this anyway!
- 1: HTML-escape data before putting it into element content
  - … **UNTRUSTED** …
  - Or any other HTML element
  - Minimum escape: & → & < → < > → > " → " ´ → ' (' is not recommended!) / → /
  - This is the standard and typical example!

Cross-site-scripting: Prevention

- 2: Attribute-escape data before putting it into “normal” attributes
  - Does not apply to href, src, style, event handlers → Rule 3!
  - Double quoted: <div attr=“ … **UNTRUSTED** … ”>
  - Single quoted: <div attr=´ … **UNTRUSTED** … ´>
  - Unquoted: <div attr= … **UNTRUSTED** … >
    - Should **not** be used anyway!
  - What to escape:
    - All ASCII codes below 256 → &#x??; or named entity
      - Excluding alphanumeric characters (A-Z, a-z, 0-9)
      - Why this much? Because e.g. a space (and many more: % * + , -…) end an unquoted attribute!
  - Properly quoted attributes: Can only be escaped by using the same quote → Escaping would be sufficient!
    - But can you be sure that EVERY attribute is ALWAYS quoted?
    - Even when someone later changes the HTML?
    - And what about lenient browsers (see example before with ‘</script>)?
    - Also see later: What if attribute content is put somewhere else later?

Cross-site-scripting: Prevention

- 3: JavaScript-escape data before putting it in JS data values
  - Especially: href, src, style, event handlers
  - Somewhat safe are:
    - Inside quoted string: <script>alert(´… **UNTRUSTED** …´)</script>
    - Inside quoted expr.: <script>x="… **UNTRUSTED** …")</script>
    - Inside quoted event handler:

  - Attention: Some functions are never safe (see before)
    - What takes a string and produces code from it/executes it
  - What to escape: See Rule 2 above!
    - All ASCII codes below 256 → &#x??; or named entity
      - Excluding alphanumeric characters (A-Z, a-z, 0-9)
    - Do not use "\" to escape: The HTML parser runs before the script parser and may match it (="claim as its own and so remove it")
  - All attributes should always be quoted anyway

Cross-site-scripting: Prevention

- 4: CSS-escape data before putting it into style values
  - <style> selector { property : … **UNTRUSTED** …; } </style>
  - <style> selector { property : "… **UNTRUSTED** …"; } </style>
  - <div style=property : … **UNTRUSTED** …;> text
  - <div style=property : "… **UNTRUSTED** …";> text
  - What to escape: See Rule 2 above!
    - All ASCII codes below 256 → &#x??; or named entity
      - Excluding alphanumeric characters (A-Z, a-z, 0-9)
    - Do not use "\" to escape: The HTML parser runs before the script parser and may match it (="claim as its own and so remove it")
    - </style> may close the style block even when inside a quoted string, as the HTML parser runs before the JS parser!
      - Similar to the '</script> example above
  - All attributes should always be quoted

Cross-site-scripting: Prevention

- 5: URL-escape data before putting it into URL parameters
  - link
  - What to escape: See Rule 2 above!
    - All ASCII codes below 256 → &#x??; or named entity
      - Excluding alphanumeric characters (A-Z, a-z, 0-9)
    - Entity encoding is completely useless here!
- Attention: This does NOT apply to whole URLs
  - Neither absolute nor relative ones!
  - Such URLs must be encoded according to where they appear, e.g. as attribute values
    - link → Attribute-escaping
    - Also make sure to check the protocol
    - Should also check, that no unwanted parameters are in there
      - E.g. encoded JavaScript, unique IDs (→ privacy), …

Cross-site-scripting: Prevention

- Additional rule: these rules are only as “simple”, if the content remains where it is inserted. If JavaScript later takes it and puts in a different context, it needs to be escaped **again** for the target context!
- Example (note escaping of ‘ to ' !):
  - ```
    var string='<img src='b'
    onerror=alert(6);>';
    document.getElementById("target").innerHTML=string;
    ```
  - No problem initially – the string is nicely escaped for putting it into a quoted string literal
  - But when inserted into document, it suddenly becomes dangerous
  - Reason:
    - ‘<‘ is (mostly) harmless as text content of a quoted string
      - See examples before…
    - ‘<‘ is **NOT** harmless at all as direct element content!
    ```

XSS: Prevention summary

- Always quote all attributes
  - Properly escape all content in it, especially the quotes!
- Do not put user-supplied data into dangerous areas
  - Tag content and attribute values: Often unavoidable
  - JavaScript code: Should not be necessary!
  - CSS: Should not be necessary!
  - URL parameters: Should not be necessary!
  - Any other place: Never ever!
- Use checked, verified, and tested libraries for escaping
  - Writing them is not trivial (but not that complex either…)
- Use CSP, policy engines, frameworks etc if available
- Take special care with your JavaScript code
  - What happens when the page looks different than it should?
    - DOM-based XSS!

Cross-Channel Scripting

Cross Channel Scripting

- Similar to XSS, but involves two protocols
  - E.g. SNMP + HTTP, P2P + Webserver…
- Example:
  - Create torrent file with real content + an additional attacking file
    - Filename is actually a script, e.g. “<iframe onload=‘…’> 1.pdf”
  - Seed it to big trackers
  - Users download the file
  - When the content is shown in a webbrowser, e.g. the UI of a NAS system, this script is executed
  - The script can be external or the additional file itself
- Note:
  - The main file actually **does** contain the desired content
  - The attack might be (visually) cleverly hidden so the user does not notice anything abnormal

Cross Channel Scripting

- Also often vulnerable: Lights-out management
  - Contains a webserver; traffic to it is captured before the OS has a chance to see (and inspect, filter…) it!
  - Example: Specify the attacking script as the username on a (failed) login try; it will end up in the log. When the administrator looks at the log, the attack is executed!
- Also problematic are APIs and third-party extensions
  - Example: Facebook and Twitter
    - Facebook stores data “raw” and escapes it on output
    - Third-party apps are expected to also escape it – but what if they forgot about this? → Vulnerability!
- Detection and Prevention:
  - Similar to XSS, but more difficult
- Especially important:
  - Whatever the source of data, it must be appropriately quoted!

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