Computer Security Seminar
API Attacks

Security Engineering/Ross Andersson, Chapter 18

Shai Ziv
25th May, 2014
Application Programming Interface

- Interface for communication between two programs.
  - Two threads of the same program.
  - Two programs running on the same server.
  - Client and server.
API is Vulnerable

• Door to the outer world.
• Untrusted sources give commands.

• Designing a secure API is very difficult.

• Small programming oversights can be disastrous.
The Perfect API

void API(void)
{
    printf(“No commands available”);
}

Useful, eh?
Attack on Visa Security Module

- Hardware device for bank security.
- Stores no memory.
  - Only a single master key stored in tamper-resistant memory.
  - Encryption under this key is “unbreakable”.

- We will look at the Terminal Key Generation for ATMs
- ATM security is based on dual control (secret sharing)
  - \[ K = K_1 \oplus K_2 \]

• Key creation:

\[
\begin{align*}
\text{Program} & \xleftarrow{E_M(K_1)} \text{VSM} \xrightarrow{K_1} \text{Worker 1} \\
\text{Program} & \xleftarrow{E_M(K_2)} \text{VSM} \xrightarrow{K_2} \text{Worker 2} \\
\text{Program} & \xrightarrow{E_M(K_1), E_M(K_2)} \text{VSM} \\
\text{Program} & \xleftarrow{E_M(K = K_1 \oplus K_2)} \text{VSM}
\end{align*}
\]

• What happens if we insert the same encrypted key twice?
  • \( K = K_1 \oplus K_1 = 0 \).
  • Known key inside the system.

• The problem: Support of offline ATMs.

  • $\text{Program } \xrightarrow{E_M(PIN),E_M(K)} VSM$
  
  • $\text{Program } \xleftarrow{E_K(PIN)} VSM$

• $PIN = D_0(\cdot)$. 

• How to fix?
• Independent atomic commands!

$\text{Program} \rightarrow VSM \leftarrow E(K) \rightarrow K_1 \rightarrow \text{Worker 1} \quad K_2 \rightarrow \text{Worker 2}$
Attack on IBM PIN Generation

• Wild credit cards appear!
• IBM uses PIN generation.
• It’s not very effective...

• In IBM PIN code generation, $PIN_C$ depends on 3 values:
  • $PIN_M$ – bank’s master PIN.
  • $N_C$ – account number.
  • offset – for memorable (weak) PIN.
Attack on IBM PIN Generation – cont.

• The algorithm:

\[
\begin{align*}
Hex &= E_{PIN_M}(N_C) & Hex &= a2ce126c69aec82d \\
Dec &= Dec_Table(Hex) & Dec &= 022412626904823 \\
PIN_C &= Dec[1..4] + offset & PIN_C &= 0224 + 6565 = 6789
\end{align*}
\]

• Great Idea (?): \(Dec_Table\) is supplied by the user.
  • \(Dec_Table = 0123456789012345\) was widely used.
Attack on IBM PIN Generation – cont.

• Set $Dec_{Table} = 0000000000000000$.  
  • Get $E(PIN_c = 0000)$.

• Set $Dec_{Table} = 1000000000000000$.  
  • If $E(PIN_c)$ changed, then it contained a ‘0’.

• And so on...

• With a few dozen queries $PIN_c$ can be found.
Attack on IBM PIN Generation – cont.

• How to fix?
  • IBM’s “solution”:
    • Must contain at least 8 different characters, that appear at most 4 times.
    • What about “0123456789012345”, then “1123456789012345”, and so on?

• Be careful when using user’s input, and avoid it as much as possible.
  • Remember the perfect API!
API Programming - Input Check

• The API itself can be 100% safe.
• The communication still will not be secure.

• Before you execute, check the input you are executing!
SQL Injection (Input Check – example)

- 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 code:

```sql
select *
from workers
where name = ('$$');
```

• Expected parameter ($$): Shai Ziv

• Attacker’s parameter:

```sql
Shai Ziv');
insert into workers values ('Joffrey Baratheon
```

• When inserted:

```sql
select *
from workers
where name = ('Shai Ziv');
insert into workers
values ('Joffrey Baratheon');
```
SQL Injection – cont.

Hi, this is your son’s school. We’re having some computer trouble.

Oh, dear – did he break something? In a way –

DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

Well, we’ve lost this year’s student records. I hope you’re happy.

And I hope you’ve learned to sanitize your database inputs.
Buffer Overflow (Input Check – example)

• 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?

```c
main(
{
    char buffer[128];
    gets(buffer);
}
```
Buffer Overflow – The Stack

Stack

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

gets frame

gets frame
No Buffer Overflow – No Attack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Gets locals</th>
<th>BP</th>
<th>Return addr</th>
<th>Param - buffer</th>
<th>Buffer</th>
<th>BP</th>
<th>Return addr</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Gets locals</th>
<th>BP</th>
<th>Return addr</th>
<th>Param - buffer</th>
<th>Hello World!</th>
<th>BP</th>
<th>Return addr</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Hello World!</th>
<th>BP</th>
<th>Return addr</th>
</tr>
</thead>
</table>
Buffer Overflow – The Attack

Stack

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

Code:

- download('nyan.mp3')
- while (1)
  - play('nyan.mp3')
- ~override~
- buffer addr

• main finishes as usual
• Computer is infected
Summary – API Design

• Designing a secure set of commands is very difficult.

• Single secure looking command might be insecure.
• Multiple secure commands might be insecure when combined.
• Each user input can be used for an attack.
Summary – API Design – cont.

• Simplicity is key.
  • Complicated APIs are all the more vulnerable.
  • Atomic and independent commands.

• Many failures happen when adding features to API.
  • When designed initially, those features were not considered.
  • The feature itself should be checked and rechecked.
  • Relations between the new feature and old features might be problematic.
  • Is the feature necessary?

• Use as minimal input as possible.
  • There is no reason to use a parameter from the user, when you know its value in advance.
Summary – API Implementation

• Input check.
• Input check.
• Input check.

• The code which handles the user’s input is extremely critical, and should be treated that way.
Backup
Attack on the 4758

- 4758 is IBM’s equivalent to Visa’s module.
- The 4758 supported “check value” creation for a key $K$
  - $check = E_K(0)$

- At the time the key length was 56 bits.
  - This means we need $2^{55}$ effort to crack an unknown key, which is (not really) too much.
Attack on the 4758 – cont.

• We do not need to crack a specific key.
  • The 4758 would re-encrypt data with a different key

• Meet in the middle attack:
  1. Collect a number of check values. Say $2^{16}$ (takes a few hours)
  2. Store them in a hash table.
  3. Go over keys until you get a hit. (takes $\frac{2^{56}}{2^{16}} = 2^{40}$ effort)
  4. ??????
  5. Profit.