Web application attacks in practice

(or how the common real application attack looks like)

Ing. Pavol Lupták, CISSP, CEH

Contents

- Achieving the attacker's anonymity
- Looking for SQL/XSS injections
- SQL code obfuscation
- Cracking the hashes
- Exploiting admin web interface
- Local root escalation
- Cleaning the traces
- Backdooring

Anonymity

In order to achieve the maximum anonymity, the attacker has various choices:

- TOR – the easiest way (problem can be with bandwidth, especially in aggressive attacks)
- hacking any Internet vulnerable server (there are millions) and escalating root privileges
- using anonymous shell accounts (freeshell.eu)
- buying anonymous server accesses, anonymous proxies using stolen credit cards

The attacker has to be aware of

- sanitizing its “browser footprint” (e.g. using privoxy)
- **DNS leaks** (using TOR internal DNS resolver)
- **Traffic analysis**
- **Eavesdropping by exit nodes**
  (by **not sending any sensitive information** that can reveal the attacker's identity)

Strong anonymity can be achieved
quite easily!

Let's look for SQL/XSS injections

- port scanning can be very aggressive and easily detected
- web spidering and looking for **certain SQL/XSS injections** is much more undetectable (without WAF/IDS/IPS and using POST injections it looks like a legitimate traffic)
- it's easy to enumerate most SQL injection “vectors” (and also it's quite easy to block most of them using WAF/IPS)

Arithmetic Blind SQL injections

No need to use AND or OR operators!

- SELECT field FROM table WHERE id=abs(param)
- SELECT field FROM table WHERE id=A+ASCII(B)-C
- SELECT field FROM table WHERE id=A+(1/(ASCII(B)-C))
- B is substring(passwd,1,1), C is counter, when ASCII(B)=C, “divide by zero” exception occurs

Time-based blind SQL injections

Depends on injection query SELECT IF(expression, true, false) and it can be used in all database servers:

- **MySQL** - BENCHMARK(5000000,MD5(CHAR(116)))
- **MSSQL** – 'WAIT FOR DELAY' 0:0:10
- **PostgreSQL** - pg_sleep()
- using database dependent “**Heavy queries**”

Time-based Blind SQL injections using Heavy Queries

- the attacker has to find heavy slow queries that can be consequently used as “delay” (e.g. time-expensive cross-joining conditions)
- program.php?id=1 and (SELECT count(*) FROM sysusers AS sys1, sysusers as sys2, sysusers as sys3, sysusers AS sys4, sysusers AS sys5, sysusers AS sys6, sysusers AS sys7, sysusers AS sys8)>1 and 300>(select top 1 ascii(substring(name,1,1)) from sysusers)

Marathon Tool

- automates time-based blind SQL injection attacks using Heavy Queries in SQL server MySQL, MS Access, Oracle DB server
- extracts data using heavy queries from all these databases
- http://www.codeplex.com/marathontool

SQL injection tools

- SQL injection can be fully automatized by using many tools (sqlmap, SQL Power Injector, 0x90, SQLiX, sqlninja and many commercial ones)
- many of them support various SQL injection methods, including “arithmetic” or “time delay” blind SQL injection
- the problem can be with the old DBs without information schemas (like MySQL 4.x) where the attacker has to guess the tables names, columns

“Potential” SQL injections can be practically exploited in 99% cases!

The application is vulnerable to XSS/SQL injection, but it uses WAF/IPS

- injection payload can be obfuscated by the attacker, e.g. using Hackvertor
- many WAFs can be evaded by this technique (including mod_security, PHP IDS and many commercial WAFs)

mod_security & PHP IDS bypass

Special UTF encoding, malformed hex entities & other tricks are used to bypass WAFs:

- **mod_security bypass**

```
<div/style=`-:expressio&#x5c&#x36&#x65(\u006&#x34;omai&#x6e=x)` x=modsecurity.org>
```

- **PHPIDS bypass**

```
<div/style=-=expressio&#x5c&#x36&#x65(-execScript(x,y)-x) x=MsgBox-1 y=vbs>
```

WAFs are just workarounds!

No WAF/IPS can 100% protect you from input validation attacks (even if your WAF vendor claims it :-)

The attacker's interests

The attacker is **mainly interested in**:

- any sensitive information (credit card numbers, usernames, passwords, certificates, keys, ..)
- administrator/root passwords / hashes (in order to escalate his privileges, access to admin web interface or gain full database access)

Cracking the hashes

Hashes are **invaluable source** for the attacker doing cracking by using:

- big word-lists/dictionaries
- brute-forcing (this can be very time consuming)
- offline or online rainbow tables (for all hashes which do not use salt including DES, LM, NTLM, MD5, A5/1, ...)

Admin web interface

- the admin web interface is almost always much more complex than the main application, **less restrictive allowing using own SQL statements** or **running own system commands** and it is very often **available from the Internet** (!)
- the attacker cracks the admin hashes and gains the access to the administrator web interface

Another way to gain the admin web access I.

- exploits a potential XSS vulnerability in the application, the admin interface has to be in the same SOP like the main application
- it requires no admin interaction (in case of persistent XSS, just reading “XSS-infected” forum by administrator)
- it requires admin interaction (in case of reflexive or dom-based XSS), e.g. clicking “infected” link

Injected javascript functionality

- reads admin session ID from the cookie
- if the cookie uses HttpOnly flag, it performs TRACE request using XMLHTTPRequest and receives the cookie in HTTP response
- can perform CSRF request (and e.g. change admin password)
- can do a lot of ugly things (e.g. portscanning admin's intranet)

Another way to gain the admin web access II.

- exploits bad session management implementation (when the application uses just cookies for session ID), thus the application does not need to be vulnerable to SQL/XSS (!)
- the attacker can perform CSRF request (send arbitrary GET/POST request under administrator), e.g. change admin password, send forgotten password, set admin privileges to another user, etc.

Exploiting of local system

- possibility to use own SQL statements (e.g. with **execute privileges** or **reading/writing local files**) or **run own system commands** almost always leads to gaining the local system user access (apache, www-data, webuser, ..)
- consequently, exploiting the kernel is obviously easy (there are many public available local root exploits for all Linux kernels <= 2.6.31)

Care about your kernel and local system security!

(e.g. using on-the-fly kernel patching)

Cleaning the traces

- after the successful attack, web server / WAF /IPS / IDS logs are full of SQL injection attempts
- when the attacker gains local root/admin, it's usually easy to clean his traces (modify web server / WAF / IPS / IDS logs, remove suspicious entries from the database, ...)

Always store logs on the read-only medium or send them to the remote non-hackable log server!

Backdooring

- nowadays, the best way of backdooring is to use LKM rootkits – they are stealthy, non-detectable, and completely immune against file-system checksums/hashing (like Tripwire)
- compromised server can be used as another attacker proxy, or sold as a part of botnet on the hacker's blackmarkets

Summary

- security should be always “**multi-layered**”
- write secure code, use prepared statements
- use **whitelisting** instead of blacklisting
- do input and also **output** validation
- use 3rd layered database architecture
- care about your local system security and kernel

References

- http://proidea.maszyna.pl/CONFidence09/2/CONFidence2009_chema_jose.pdf
- http://www.owasp.org/index.php/Blind_SQL_Injection
- http://proidea.maszyna.pl/CONFidence09/2/CONFidence2009_gareth_heyes.pdf

Thank you for your attention!

pavol.luptak@nethemba.com