Maximizing the speed of time based SQL injection data retrieval

30c3, Hamburg, 29.12.2013

Arnim' ; DROP TABLE students;--)

Introduction SQL injection

- SQLi is #1 of OWASP Top 10 Web vulnerabilities
- Sample code of vulnerable php script:

```
$sql_cmd =“SELECT * FROM user WHERE id = ” . $_POST['id'];
```

- Form-Input : `42; UPDATE user SET type="admin" WHERE id=23;`
- Resulting Query:

```
SELECT * FROM user WHERE id =42; UPDATE user
SET type="admin" WHERE id=23;
```

Retrieving Data with Classic SQLi

```
SELECT author, subject FROM article
WHERE ID=42 UNION SELECT login,
password FROM user;
```

Very fast, can sometimes retrieve multiple strings in one request.

Boolean SQLi

- No output of the query can be seen
- There's an indication, if the result of a query is true or false because a certain string appears in the webpage, e.g. an error message
- Fastest retrieving method is binary search:
  - Is the ASCII-Code of the 1st character of the password of user 'admin' lower than 64?
  - If 'true': Is the ASCII-Code of the 1st character of the password of user 'admin' lower than 32?
  - …
- Slow: Needs 7 request per ASCII character (but can be multithreaded)

Time based SQLi

- Neither output of the query can be seen nor any indication of it's result
- Only possible way to determine the result, is to let the database SLEEP() some seconds, if the query turns out false and continue immediately if it's true.
- In other databases time intense instructions have to be executed (effectively doing a short DoS)
- Very very slow and prone to errors because of hard distinction between SLEEP() and network lag
- Multithreading difficult to impossible

Common Pitfall

> If a hammer is your only tool,
> every problem looks like a nail.

Time based SQLi binary search

- String to get: “1234”, only 4 possibilities
- Is number <= 2 ?
  - If true:
    - Is it 1?
      - If true: **found 1**
      - If false: **found 2**
  - If false:
    - Is it 3?
      - If true: **found 3**
      - If false: **found 4**
  - To get 1234 it takes 8 request

Time based SQLi linear search

- Is number = 1 ?
  - If true: f**ound 1**
  - If false:
    - Is number = 2 ?
      - If true: **found 2**
      - If false:
        - Is number = 3 ?
          - If true: **found 3**
          - If false: **found 4**

To get 1234 it takes 9 request, 6 slow + 3 fast

Time based SQLi linear search 2

Lets learn from the game show “What's my line?” aka “Heiteres Beruferaten”: some questions need to reversed to avoid the answer “no” (e.g. "Is it something other than..." or "Can I rule out...")

- Is number != 1 ?
  - If false: f**ound 1**
  - If true:
    - Is number != 2 ?
      - If false: **found 2**
      - If true:
        - Is number != 3 ?
          - If false: **found 3**
          - If true: **found 4**

To get 1234 it takes **3 slow + 6 fast request**

- Morse code was designed to use the short signals on the most common characters, dit = e
- According to Benford's law 1 is the most common number

Time based SQLi compare

- Binary search: 4x fast + 4x slow
- Linear search: 6x fast + 3x slow
- True usually returns after ~100ms, false after 1 second
- Binary search: 4,4 seconds
- **Linear search: 3,6 seconds** (and only 3x tiny DoS for DBMS without SLEEP)
- Break even for binary search on 4 choices: 1x slow = 2x fast
- Break even for binary search on 8 choices: 1x slow = 3,2x fast

ASCII Table

Dec	ASCII	Dec	ASCII	Dec	ASCII	Dec	ASCII	Dec	ASCII	Dec	ASCII	Dec	ASCII	Dec	ASCII
0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	

- Avoid using non standard functions like REGEXP or RLIKE, use ASCII-ranges

Testing speedup

- Changed sqlmap to use quick request for > on 1st and 2nd request, then on <
- Retrieve sample string: AZazme_5

Sqlmap original:

```
[12:17:18] [DEBUG] performed 71 queries in 121.29 seconds
```

Sqlmap patched:

```
[12:14:44] [DEBUG] performed 71 queries in 103.72 seconds
```

Mitigation of account stealing in the wild

- So your web app is state of the art and does:
  - Input sanitation and uses prepared statements
  - Salts and hashes passwords with 50.000 rounds of PBKDF2 (more would be subject to an easy DoS by multiple authentication attempts)
  - Enforces passwords with 10 digits containing upper, lower, number, symbol + no dictionary words
- What happens:
  - A SQLi-vuln gets introduced with a new feature or a lousy plugin (check out exploit-db ;)
  - Attacker dumps hashes and cracks 10-30% of them on his GPU-cluster/cloud service in a week (*Passw0rd1!* isn't in your 100.000 word dictionary but in his containing 100.000.000 entries)

Solution: Learn from Adobes one good example!

- 150 million password-tokens leaked and not a single one got cracked (though some got guessed by the password hints)
- Encryption of OS-Passwords is considered stupid because if you can read e.g. /etc/shadow, you can read the file containing the key as well
- Store the key outside the database, unreachable for SQLi. Attacker would need a 2nd vulnerability to get it
- Passwords should be salted, hashed AND encrypted
- That's 4-6 additional lines of code
- Cracking 3DES or AES is much harder than password cracking (168 to 256 bits of entropy vs 80 of a typical password makes it 1.000.000.000.000.000.000.000.000.000.000+ times harder to crack)

Solution: Learn from Adobes one good example!

- Protects even the lousiest password, 123456 is safe (from offline cracking ...)
- Known cleartext doesn't help in cracking because of the salt
- Prevents using SQLi to overwrite existing passwords (e.g. admins) or insert new ones
- Protects passwords also from attacks which get a direct db connection on TCP 3306, 1526, ... or get access to the db backups
- Encryption beats “peppering” and keyed HMACs in flexibility for combining user databases from multiple systems because decrypt/encrypt is possible
- Allows to encrypt other sensitive data like password hints as well ;)

Bonus slide: Key management for webapps

- Take it easy, even just setting $key=... in config.php is much safer than just hashing passwords
- If you need more security:
  - Local file inclusion (LFI) and other methods to read arbitrary files on the webserver are the dangers
  - Store key in file with fixed prefix and random end, e.g. secret_key_e2e4dEAdheAd30c3.txt because LFI mostly can only read a known name and not search for files
  - Remove OS read permissions on file after reading it, e.g. `chmod 000 secret_key_e2e4dEAdheAd30c3.txt`
  - Attacker needs remote code execution to read the key file

Bonus slide: Methods for speedup

- Start with >96? instead of 64 (?)
- On ASCII-values < 32 make a lucky guess on 0 aka end of string
- Predict length of string depending on previous results, e.g. the last 3 hashes where 40 bytes long => no need to use several request to check for 0x00
- Adopt to charsets of different columns:
  - Hashes => hex = just 16 possibilities (Close gap in ASCII between numbers and lowercase chars with `SELECT @a:=ORD('a'), IF(@a > 96 AND @a <=103, @a-39, @a);`
  - Emails => make string lowercase and don't ask for uppercase with `SELECT @a:=ORD('A'), IF(@a > 96 AND @a <=122, @a-32, @a);`
- Ask for ranges instead of just greater/smaller, e.g. is the ASCII-value between 48 and 57 (=a number) ?

Bonus slide: methods for speedup 2

- Predict next chars:
  - htt => p
  - q => u
  - lengt => h
  - .ed => u
- Use network QoS settings on client and router to minimize network lag
- If there's a choice, pick fastest injection point (there might be different amounts of webapp- and SQL-code executed)