

## **Table of Contents:**

### **Web**

<b>Basic Auth.....</b>	<b>2</b>
<b>Fun with Shopping Carts.....</b>	<b>3</b>
<b>Crack Bob's Password.....</b>	<b>5</b>
<b>IOA Mail.....</b>	<b>8</b>

### **Source Review**

<b>GetLine.....</b>	<b>10</b>
<b>Webserver.....</b>	<b>11</b>

### **Crypto**

<b>SSIC'S Encryption.....</b>	<b>13</b>
-------------------------------	-----------

## Web100 – BasicAuth

**Location:** <http://18.236.83.248:32611/web100/>

**Description:** Provide the correct username to log in and find the flag. Upon starting the challenge, the page is displayed, showing a username input field and submission button.

### Methodology:

#### Step 1: Determine how Authentication occurs

Initially, I attempted to supply any username or no username to find any clues to determine how user authentication is handled by this page. An incorrect submission resulted in an alert box with an incorrect username error. This indicated that authentication is handled on the client side through Javascript. For authentication via Javascript, the Javascript files must be accessible by the user's browser. These files can be found through viewing the page source or using inspect element, so that was my next step.

#### Step 2: Locating and viewing Javascript files

Upon inspecting Web100's source, the javascript file "kernel.js" is displayed in the page head.

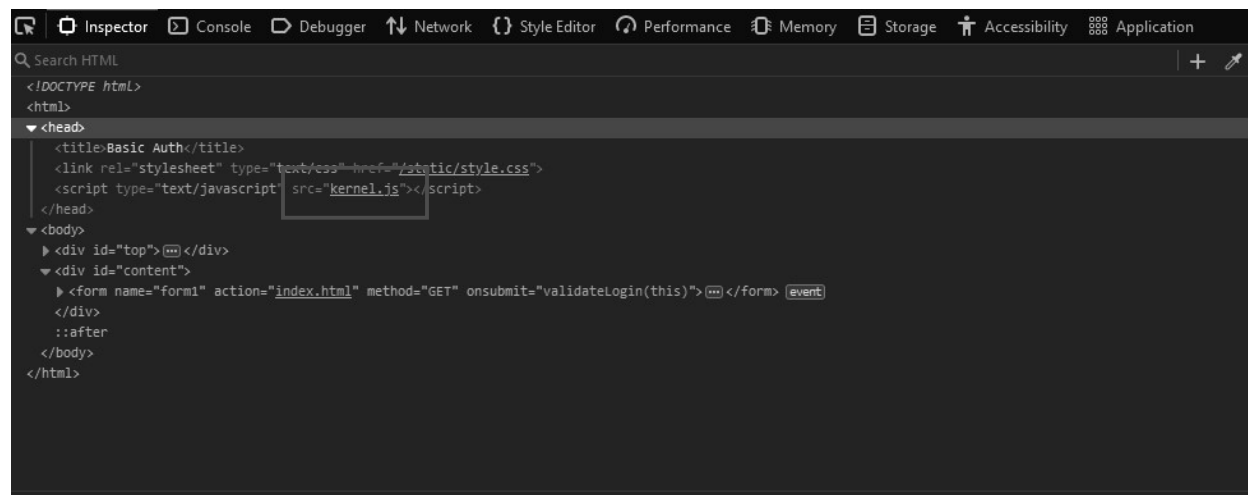


Figure 1: Finding JS file in Web100's source HTML

Navigating to Web100/kernel.js in the browser results in an empty page but inspecting Kernel.js' source resulted in interesting output. The source code reveals several Javascript functions and elements that are clearly used in user authentication, including a suspicious string "gr4nt3dUsr".

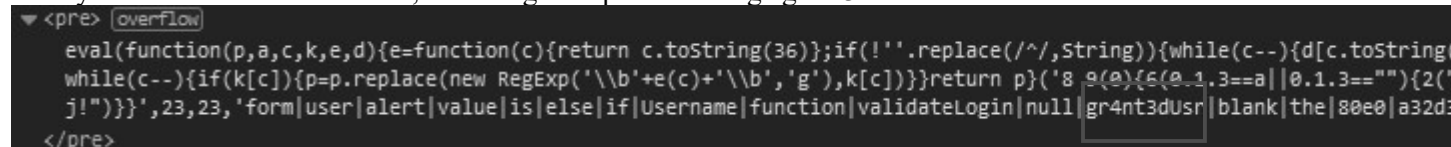
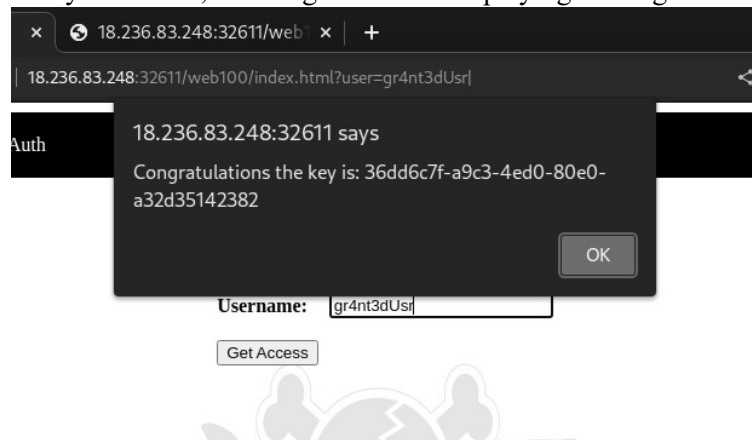


Figure 2: Suspect Username within Kernel.js source

### Step 3: PROFIT

With what I believed to be the correct username, I returned to the Web100 page and entered gr4nt3dUsr as my username, resulting in an alert displaying the flag.



**Figure 3: Entering username and discovering the flag**

#### Mitigations:

This attack was made possible by exploiting CWE-603: Use of Client-Side Authentication. All an attacker needs do is find the credentials in the page's source code and enter them. To mitigate this vulnerability, usernames and passwords should be stored in a secure configuration file, or ideally, a secured database. Authentication should always take place on the server side.

## Web150- – Fun with Shopping Carts

**Location:** <http://18.236.83.248:32611/web150/>

**Description:** Purchase the MagicFile and receive the token. Upon starting this challenge, the user is shown a shopping cart style webpage with form fields for Quantity, Price, Discount, Amount and Total.

#### Methodology:

##### Step 1: Determine what permissions users have

Initially, I attempted to modify the values in the various form fields, only to find they are read-only. I then attempted to checkout despite having a credit of \$0, which was rejected, as was expected. Being able to modify the values submitted to the application was believed to be the most likely way to purchase the MagicFile.

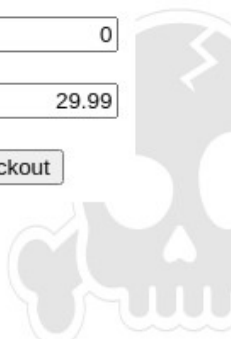
##### Step 2: Change submitted values to allow purchase

To perform the value modifications, I opted to use the Burp Proxy to intercept requests before they were submitted. OWASP ZAP can also be used to perform the same actions. The proxies can be set to function

with any browser and provide their own preconfigured browsers. For this demonstration, the browser used was Burp Suite's Browser.

Credit \$0				
Item	Quantity	Price	Discount	Amount
MagicFile	<input type="text" value="1"/>	<input type="text" value="29.99"/>	<input type="text" value="0"/>	<input type="text" value="29.99"/>
			<b>Discount:</b>	<input type="text" value="0"/>
			<b>Total:</b>	<input type="text" value="29.99"/>
Please enter your special discount code: <input type="text"/>			<input type="button" value="Apply Code"/>	<input type="button" value="Checkout"/>

Checking out...



**Figure 4: Submitting an unmodified request**

Upon clicking checkout, the Burp Interceptor displays our POST request before it can be submitted to the application, allowing us to modify the request before it is sent.

```

1 POST /web150/cgi-bin/cart.py HTTP/1.1
2 Host: 18.236.83.248:32611
3 Content-Length: 45
4 Authorization: Basic cGxheWVyMTpQcmV2YW1sNC0yVW5zZWFsZWQ0LTJMdW5jaGVvbjQtM1N1YnByaW1lNC0yRGVmbGF0b3I0LTJTbGljZWQ=
5 Accept: application/json, text/javascript, */*; q=0.01
6 X-Requested-With: XMLHttpRequest
7 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.5481.78 Safari/537.36
8 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
9 Origin: http://18.236.83.248:32611
0 Referer: http://18.236.83.248:32611/web150/
1 Accept-Encoding: gzip, deflate
2 Accept-Language: en-US,en;q=0.9
3 Cookie: session=1526403344
4 Connection: close
5
6 {
  "action": "checkout",
  "total": "29.99",
  "d": "0"
}

```

**Figure 5: Burp Interceptor displaying POST request**

As can be seen in Figure 5, values for total and discount are being submitted to the application. Changing the discount to a higher value and/or changing the total to a lower value should produce the results we

want. I changed “total” and “d” to 1. Though not displayed on the final form, this approach was successful, as shown in figure 6.

## My Shopping Cart

Credit \$0				
Item	Quantity	Price	Discount	Amount
MagicFile	<input type="text" value="1"/>	<input type="text" value="29.99"/>	<input type="text" value="0"/>	<input type="text" value="29.99"/>
			<b>Discount:</b>	<input type="text" value="0"/>
			<b>Total:</b>	<input type="text" value="29.99"/>
Please enter your special discount code: <input type="text"/>			<input type="button" value="Apply Code"/>	<input type="button" value="Checkout"/>

Congratulations on your purchase, the magic key is: c884952e-37bc-46ab-a7bf-00ca802fce5a

**Figure 6: Success!**

**Mitigations:** To mitigate the vulnerabilities exploited in this attack, care should be taken to prevent user modification of assumed-immutable data, such as the price and discount values. These values should not be submitted in a POST request as constants, rather as variables whose values are stored server side.

## Web200- – Crack Bob’s Password

**Location:** <http://18.236.83.248:32611/web200/login.html/>

**Description:** Bob always selects weak passwords, crack his without using Brute Force, as you only have three attempts. Upon starting this mission, users are taken to Web200/login.html, and username/password input fields are displayed.

**Methodology:**

### Step 1: Find Bob’s password

Since Brute Forcing the password was not an option, I needed Bob’s password. My first step was to attempt to log in with a blatantly incorrect username and password to see if any clues could be had from the resulting error message. This resulted in being taken to the page Web200/login.php, displaying the message “Invalid Username/Password”. The login.php page indicated that validation was being done server side but didn’t provide any other useful clues.

Returning to the login page, a few different PHP and SQL injections were used, attempting to dump passwords, with no success. I decided to utilize OWASP ZAP to perform a crawl of the page to find

hidden directories or an admin panel. No hidden directories were discovered but Forced Browsing to Web200 found what we were looking for, and much more.

## Step 2: Explore Web200

Navigating to Web200 results in the page shown in the following index page.

### Index of /web200/

Name↓	Last Modified:	Size:	Type:
../		-	Directory
creds.php	2022-Sep-07 07:08:56	1.4K	application/octet-stream
creds.php.bk	2022-Sep-07 07:08:56	1.3K	application/octet-stream
ioa.png	2022-Sep-07 07:08:56	7.1K	image/png
jquery.js	2022-Sep-07 07:08:56	84.6K	text/javascript
login.html	2022-Sep-07 07:08:56	0.8K	text/html
login.php	2022-Sep-07 07:08:56	0.6K	application/octet-stream
passwd	2022-Sep-07 07:08:56	0.1K	application/octet-stream
style.css	2022-Sep-07 07:08:56	0.4K	text/css

lighttpd/1.4.63

**Figure 7: Web 200 Index Page**

Several filenames stood out immediately, namely passwd and creds.php. Opening creds.php resulted in a blank page, but opening creds.php.bk resulted in the script used to perform user validation, password hashing and the two salts used in the hashing process.

```

$LEADING_SALT = "soMe3FK2J+YFAMzUjwpfpA";
$TRAILING_SALT = "6dcKXY3x/JRZbyj9RDB/Lw";

function is_valid_user($username, $password) {
    global $LEADING_SALT, $TRAILING_SALT;
    $pwd_file = fopen("passwd", "r");
    $result = false;

    if($pwd_file) {
        while (($line = fgets($pwd_file))) {
            $comps = explode(":", trim($line));
            if (count($comps) == 2 && $comps[0] == $username) {
                $hash = md5($LEADING_SALT . $password . $TRAILING_SALT);
                if ($hash == $comps[1]) {
                    $result = true;
                }
                break;
            }
        }
    }
}

```

**Figure 8: creds.php.bk showing salts and hashing algorithm**

With the information provided by creds.php.bk, all that is needed now is Bob's password hash, which can be found in the passwd file as follows: bob:b5406d12c28fe602b27b063a2c1231f9. Now, all that needs to be done is to test potential passwords against Bob's password hash following the hashing process used in creds.php.bk.

### Step 3: Finding Bob's Password

To find Bob's password, I used a BASH script that iterated through the wordlist "Rockyou.txt", a dictionary of weak passwords. Each line of Rockyou was combined with the leading and trailing salts, and then hashed using the MD5 algorithm, and then finally placed into a "hashes.txt" file.

```
#!/bin/bash
salt1="soMe3FK2J+YFAMzUjwpfpA"
salt2="6dcKXY3x/JRZbyj9RDB/Lw"
i=0
cat "$@" | while read -r line; do
    printf %s "$salt1$line$salt2" | md5sum | cut -f1 -d' ' >>hashes.txt
done
echo "DONE"
```

**Figure 9: BASH script to hash passwords**

Once hashed, I searched the hashes.txt file for Bob's hash, finding it on line 1547, corresponding the plaintext "skippy".

1546	7fe6f078c98525e2d059e532fff87166	1546	070707
1547	b5406d12c28fe602b27b063a2c1231f9	1547	skippy
1548	b4f65dad8d3ff5873f46142e38681ca6	1548	kaykay

**Figure 10: Bob's password hash and corresponding plaintext password.**

Entering Bob/skippy into the username/password fields in Web200/login.html resulted in receiving the flag, shown in figure 11.



**Figure 11: Successful login!**

**Mitigations:** Accessing this flag required a string of vulnerabilities, starting with the improper authorization of the index. The index page should be off limits to unauthorized users, as it contains everything needed to crack Bob's password. Password hashes should also be stored in a more secure area than a page index, such as a secure database.

Next, the use of an obsolete hashing algorithm with MD5 should be avoided. A dictionary-based attack was used in this example, however, MD5 is considered insecure for several reasons. It can be brute forced relatively quickly and the algorithm has been shown to have collisions. A collision is when two words produce the same hash, meaning an attacker may not even need to crack the correct password to access the system.

Finally, strong passwords should be enforced. An all-lower case, six-character password can be brute forced almost instantly. A combination of letters, numbers and symbols should be used. Password length requirements should also be enforced.

## Web250- – IOAMail

**Location:** <http://18.236.83.248:32611/web250>

**Description:** IOA has a custom email client, the flag can be found in an Admin's Email Spool. Upon starting this mission, the user encounters a login page requesting a username and password.

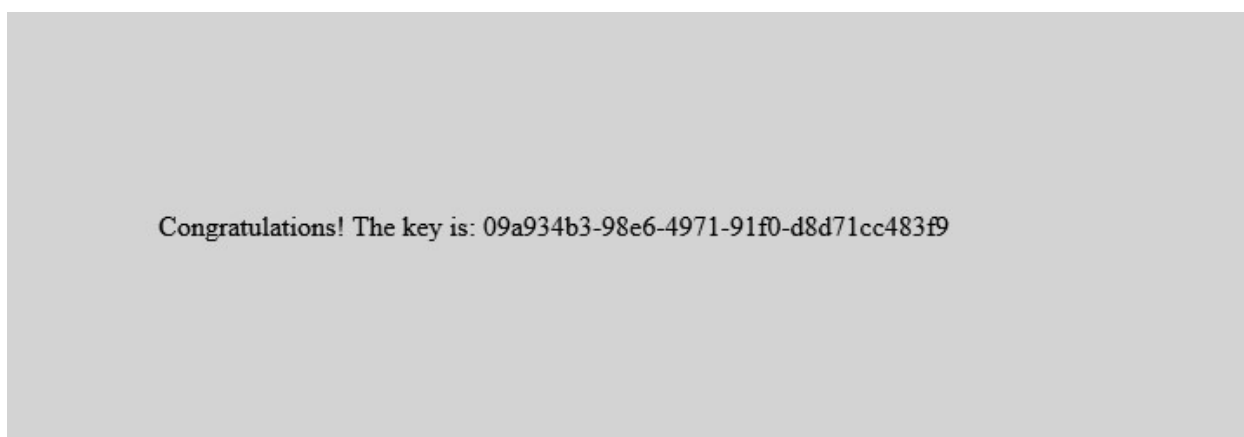
### **Methodology:**

I used a similar process here initially as I did when looking for information on Bob's Password in the last challenge. I entered a test username/password to see if any error messages would provide clues to how to retrieve the key. This produced nothing interesting. There was no index with password hashes or any other noticeable page with information about the admin's account, so I decided to see if the page was vulnerable to PHP or SQL injections in order to dump admin credentials or any other useful information, instead I was instantly treated to the flag.





**Figure 12: A simple SQL Injection just to determine if the vulnerability exists**



**Figure 13: The vulnerability does exist**

**Mitigations:** Injection based vulnerabilities are one of the most common attack vectors. Injection has been near the top of the OWASP top 10 Vulnerabilities for years. These vulnerabilities occur when user supplied input is not validated and sanitized properly. Hostile data is then used to directly supply an SQL query. The result is that the attacker can escape the intended SQL statement and supply their own statement, as was done when I supplied ' OR 1=1 --'. This resulted in SQL returning the desired output if either the appropriate username and password were entered, or the user supplied statement. Since 1 is always equal to 1, the flag was returned.

To prevent injection attacks, inputs should be sanitized, and queries should be parameterized. A query such as `"SELECT * FROM users WHERE email = '" + email + '"` is vulnerable to injection, and should be avoided. Instead, a query such as `String sql = "SELECT * FROM users WHERE email = ?"`; followed by the user's input in a separate prepared statement, and then executing the query once the parameters from the user's input are set.

## Src200- – GetLine

**Location:** <http://18.236.83.248:32611/src200>

**Description:** GetLine is a C program that takes input from a file and prints that input to the terminal.

### Methodology:

#### Understanding the Code

Since the Getline program is contained entirely in one source file, I opted to manually review it as the first step, to understand what the code is doing and to highlight any vulnerabilities that presented themselves. Upon reviewing the code, I noticed two potential vulnerabilities within getLine, a Buffer Overflow vulnerability and a Directory Traversal vulnerability.

#### Understanding Buffer Overflows

On line 36, we see the code “char buff[1024];”, Creating a 1024 character array to serve as our input buffer, when getLine is run, it can handle up to 1 Kilobyte of input, as each character is 1 byte in size. Any input that exceeds 1 KB will result in an overflow, dumping that input into the surrounding memory. This can crash the program or cause it to run in unintended ways. Consider a program designed to authenticate a user through a password. If the password is vulnerable to an overflow, it would be possible to authenticate despite an incorrect password being entered.

At its most dangerous, malicious code can be included into input in a way that code will then be inserted into memory, circumventing in place security policies. This vulnerability can impact Data Confidentiality, Integrity and Availability, also known as the CIA triad.

- Confidentiality – Buffer Overflows can be utilized to access data in memory that would otherwise be off limits to the user/program.
- Integrity – Similar to Confidentiality, Buffer Overflows can be used to place new data into memory, modifying or completely overwriting previous data.
- Availability – Overflows can lead to program or system crashes.

#### Mitigations:

Buffer Overflows are a common vulnerability in the C programs. The C language does not have many built in protections against poorly written code. There are multiple methods to prevent this vulnerability in a program, such as:

- Language Selection: As previously mentioned, C does not have many native protections built into it, where possible, it may be prudent to use a language with built in safeguards such as Java or C#, which also provide memory safeguards by running on top of a virtual environment.
- Secure Coding Practices: If C, or another language vulnerable to Buffer Overflows must be used, perform buffer checks in code to ensure appropriate buffer size. Also validate inputs to ensure that the input length does not exceed the buffer size.
- Use copy functions that accept length arguments to prevent copying more than the buffer will allow.

#### Further Reading:

For more information on Buffer Overflows, please refer to <https://cwe.mitre.org/data/definitions/120.html>

## Understanding Directory Traversal

Directory Traversal is a vulnerability that can potentially expose sensitive data when a program opens a file on the system specified by the user. If a user is allowed to supply a path that is not within the local directory, he may be able to read or modify files that he does not have permissions for.

For example, the logical use of `getLine` would be “`getLine (localfile.txt)`”, thus opening the file in the local directory. However, if the user supplies “`getLine (../../etc/passwd)`” `getLine` will output the first line of the `passwd` file.

This vulnerability can impact Data Confidentiality, as an attacker may be able to expose sensitive data through use of the `getLine` program.

### Mitigations:

- Input Validation: Exclude directory separators such as “/” or “\”, also only allow single instances of “.” To prevent accessing parent directories.
- Restrict file input to files from the local directory.

### Further Reading:

For more information regarding Directory Traversals, please refer to <https://cwe.mitre.org/data/definitions/22.html>

## Src250- – WebServer

**Location:** <http://18.236.83.248:32611/src250>

**Executive Summary:** Most of the vulnerabilities discovered in the WebServer Source code were the result of using outdated functions that do not enforce max input sizes to avoid overflows. If a secure alternative exists, it has been suggested in the mitigations section of that vulnerability. Otherwise, programmers should take care to perform checks within the code to enforce max input sizes.

**Description:** Webserver written in C using socket programming. Program is made up of eight C files, `main.c`, `args.c`, `clients.c`, `handlers.c`, `http.c`, `log.c`, `mimetype.c` and `str_util.c` and seven corresponding header files.

### Methodology:

The code of the WebServer is more complex than the `getLine` program, so I utilized two opensource SCA tools design for C and C++ code to aid in determining where, if any security vulnerabilities existed in the source files for WebServer. The Analyzers used were `flawfinder` and `Visualcodegrepper`. I then referred to MITRE’s CWE pages for information about unfamiliar C functions to help determine vulnerability status.

The vulnerabilities discovered were as follows:

### Clients.c:

**Strlen function:** On lines 155, 161 and 193, `strlen` is called in the `send_html_escape` function. `Strlen` returns a string’s length and returns an integer value. `Strlen` is an unbounded function, if it is passed a

string that is not null terminated, it will continue to read until it finds the terminator character, resulting in an overflow. This can potentially impact data confidentiality, as out of bounds memory is read. The use of `strlen` is considered deprecated, and is banned by multiple organizations, most notably Microsoft.

[Microsoft Banned Functions](#)

### **Mitigation:**

The use of `strlen` should be replaced with a function that enforces a maximum length to prevent overreads. The function `strnlen_s` is suggested as a replacement, it can be found in the Safe C String Library (SafeStr).

### **Handlers.c**

#### **Sprintf function:**

On line 133, the `sprintf` function is called in the `handle_get` function. `Sprintf` sends output to a string when passed a string pointer. `Sprintf` uses no safeguards to ensure the output of the function will not exceed the buffer size, resulting in a potential buffer overflow. Please see the Buffer Overflow segment of the `getLine` section for further information on Buffer Overflows. Notably, `sprintf` is also on the Microsoft Banned Functions list.

### **Mitigation:**

Use a function that ensures output cannot exceed buffer size. The `snprintf` function should be used in place of `sprintf`. `Snprintf` requires a size argument that prevents writing beyond buffer limits.

#### **Strlen Function:**

On line 144, the `strlen` function is called, please refer to the `strlen()` risks and mitigations in `Clients.c` section above.

### **Http.c**

#### **Strlen Function:**

On line 35, the `strlen` function is called, please refer to the `strlen()` risks and mitigations in `Clients.c` section above.

### **Args.c**

#### **Getopt function:**

On line 17, `getopt` is called, the `getopt` function is vulnerable to Buffer Overflows if the supplied `argc` and `argv` values exceed `MAX_SAFE_ARGC` or `MAX_SAFE_ARGLLEN` values respectively. Please see the Buffer Overflow segment of the `getLine` section for further information on Buffer Overflows.

### **Mitigation:**

Before calling `getopt`, `argc` and `argv` values should be checked and ensured that they do not exceed the max values.

**Atoi function:** On line 20, `atoi` is called. This function converts a string into an integer value. If a string cannot be converted into an integer, it can create an integer overflow, which has similar potential impacts

as Buffer Overflows.

### Mitigation:

In place of `atoi()`, `strtoi` should be used, which returns a max integer, should an overflow occur, `strtoi` returns a max long.

## Crypto 400- – Super Secure ICS Encryption

**Location:** <http://18.236.83.248:32611/crypto400>

**Description:** Given the ICS encryption algorithm, packet structure, six plaintext transmissions and 12 encrypted transmissions, crack the eight-character password used to encrypt message traffic.

### Methodology:

This flag was a bit of a challenge, as I initially misunderstood the encryption algorithm and wasted too much time comparing ciphertexts with the elements of the plaintext messages that appear to be static. As the first five bytes don't change, or change every other message, the plan was to perform XOR with one of the ciphertexts and the plaintext messages to see if any useful partial keys would present themselves. Every 'key' discovered using this method fell well outside of the realm of typable ASCII characters and were of no use.

Reviewing the encryption algorithm a little more closely, I realized my mistake and considered how exactly the ICS crypto functioned. Starting at the end of the message, the algorithm XORs the last message segment with the key byte pertinent to the message segment, for example, on an ACK message with seven bytes, the final message will be encrypted with the 7<sup>th</sup> byte of the key. The  $i$  variable then decrements, and the plaintext byte  $P_i$  is XORd with the key byte  $K_i$  and the previously encrypted ciphertext  $C_{i+1}$ , resulting in the ciphertext  $C_i$ . This process is repeated until  $i$  reaches 0.

$$C_{\text{final}} = P_{\text{final}} \oplus K_{\text{final}} \rightarrow C_i = P_i \oplus K_i \oplus C_{i+1}$$

### Reversing the Process:

XOR is a simple binary mathematical process but is powerful despite its simplicity. When XOR takes place, two binary values are compared, and where they share the same value, the corresponding value in the subsequent binary is 0. For example, if two hex values of 0xF, representing 1111, were XORd against each other, the result would be 0x0, or 0000 in binary. Where values are not alike, the new binary has a value of 1.

Just knowing the algorithm used to encrypt data isn't generally enough to crack it, in fact, most of the algorithms used to secure the internet and other vital data are well known. In a symmetric key system, one where the same key is used to encrypt and decrypt information, such as this one, key length is critical. Short keys, similar to passwords, are subject to brute forcing.

For this exercise, brute force was not necessary, as several plaintexts were provided alongside the ciphertexts, as shown below.

38 Eb 8E 09 00 80 04 00 00 ab 00	ab f4 77 ad cd 99 7c 0c 64 2a 15
38 Eb 8E 89 00 15 6d	3e 61 e2 38 d8 8c bb
38 Eb 8E 09 00 80 05 ff ff 81 3f	a7 f8 7b a1 c1 95 70 02 6a 24 75
38 Eb 8E 89 00 15 6d	3e 61 e2 38 d8 8c bb
38 Eb 8E 09 00 80 05 2c 78 20 24	b5 ea 69 b3 d3 87 62 17 7f 31 e5
38 Eb 8E 89 00 15 6d	38 67 e4 3f df 8b 8b
	b3 ec 6f b5 d5 81 64 10 78 36 d5
	3e 61 e2 38 d8 8c bb
	0e 51 d2 08 68 3c d9 ad d8 05 20
	3e 61 e2 38 d8 8c bb
	4a 15 96 4c 2c 78 9d ea 9f 1b ac
	3e 61 e2 38 d8 8c bb

**Figure 14: Plain text(L) and Ciphertexts(R)**

As you can see, in every plaintext message, the first three columns never change. While there is no guarantee that will be the case for every message, if it holds true, it significantly reduces the amount of time and effort required to crack the key. Before committing to writing any code to automate this process, I wanted to test my theory about the static values in the plaintext. Given the algorithm as described previously, the problem can be represented as  $0x38 = 0xAB \text{ xor } 0xF4 \text{ xor key1}$ , which is the equivalent of  $\text{key1} = 0xAB \text{ xor } 0xF4 \text{ xor } 0x38 = 0x67$ , or the letter g. This process was repeated with columns 2 and 3, once it was determined that this process would work, I wrote a python program to finish the job. The password was revealed to be “ghTiTeth”, and the resulting decrypted messages are as follows.

```
Cracked Encryption key:
g h T i T e t h
Decrypted output:
Update
0x38 0xeb 0x8e 0x9 0x0 0x80 0x4 0xf 0x26 0x6b 0x42
ack
0x38 0xeb 0x8e 0x89 0x0 0x52 0x68
Update
0x38 0xeb 0x8e 0x9 0x0 0x80 0x6 0xf 0x26 0x5 0x22
Ack
0x38 0xeb 0x8e 0x89 0x0 0x52 0x7a
Update
0x38 0xeb 0x8e 0x9 0x0 0x80 0x1 0xf 0x26 0x80 0xb4
Ack
0x38 0xeb 0x8f 0x89 0x0 0x65 0x4c
Update
0x38 0xeb 0x8e 0x9 0x0 0x80 0x0 0xf 0x26 0xb7 0x82
Ack
0x38 0xeb 0x8e 0x89 0x0 0x52 0xc1
Update
0x38 0xeb 0x8e 0x9 0x0 0x80 0x0 0x12 0xb5 0x71 0x77
Ack
0x38 0xeb 0x8e 0x89 0x0 0x52 0x85
Update
0x38 0xeb 0x8e 0x9 0x0 0x80 0x3 0x12 0xec 0xe3 0xfb
Ack
0x38 0xeb 0x8e 0x89 0x0 0x52 0xcf
```

### Mitigations:

Strong cryptography keys should be implemented where possible. It may have been considered that a transmitter that broadcasts messages in hex was unlikely to be compromised, security through obscurity is not a safe strategy. Even with weak crypto, however, without the messages sent in the clear to establish a pattern, the process likely would have taken much longer. It likely isn't practical for a device to never send information in the clear, as there may be times that it must transmit plaintext, such as after a system restart. If unencrypted transmissions cannot be avoided, consider implementing a decoy protocol. For example, if a system must transmit unsecured data for a brief period, it could transmit other, meaningless data to obfuscate any patterns in the plaintext data.