Security Analysis of Patient Portals

David Pineiro, Victor Lopez, Alexandra Erixxon

May 11th, 2016

1 Abstract

Our project was to perform a security analysis of the patient portal of a major healthcare provider in the Northeast. We created a security policy summarizing the possible actions users should have as a guide and context for our attacks. Using a list of common exploits provided by OWASP, open web application security project, our group was able to find several vulnerabilities in the patient portal.

2 Introduction

Healthcare organizations are increasingly more susceptible to cyber attacks after connecting their systems to the rapidly expanding Internet [1]. They have also incorporated patient portals, a tool that allows patients to remotely access information regarding their medical history, request appointments, prescription medication, and update their allergies. Given the sensitivity associated with this information, the consequences from an adversary altering this delicate information are serious and even life-threatening.

After contacting various healthcare providers that supply patient portals, our group partnered with a prominent healthcare provider in the Northeast that wished to remain anonymous, and will be referenced hereafter as The Organization. Because our group was going to be performing extensive testing on their platform, and they support a large number of clients, they provided us with a staged system for us to run our tests against. The staged system was very limited and did not include the ability to request appointments or medications after being logged in, but it did supply us with the overall system skeleton. Our report provides an overview for the security policy enforced by The Organization, how our group organized the testing performed, and the vulnerabilities found in the system along with ways of how to eradicate these.

3 Security Policy

In order to best understand the security requirements of a patient portal, we found it necessary to break down the actors and permissible actions. We began

by determining which actors would be relevant to the portals and we have produced the following list:

- Patients
- Hospital Staff
- Outsiders
- Healthcare Provider Staff

Each of these actors has a different role in the system. We have listed out the sets of permissible actions that each actor is allowed to take within the system. As we present the exploits found in the system, we hope that the context provided here helps the reader understand why each exploit would violate the goals of a patient portal.

- Patients
  - View and manage upcoming appointments
  - Maintain their health information (e.g. allergies, immunizations)
  - View results of their medical examinations
  - Check lab results, reports, and letters from their doctors
  - Contact their doctors with medical inquiries
  - Renew their prescriptions
  - Pay any hospital bills
- Hospital Staff
  - View their upcoming appointments
  - Maintain health information of their patients
  - Approve any changes to health record
  - Submit test results for their patients
  - Contact their patient
  - Pay any hospital bills
- Outsiders
  - Opt in to using the system
  - Contact Healthcare Provider Staff for questions
  - No other special permission
- The Organization Staff
  - Virtually all actions relating to interactions between patient and doctor

4 Testing Methodology

Patient portals serve as a membrane between the system that outsiders are exposed to and the system that patients, after logging in, are exposed to. Because an adversary could play the role of an outsider and of a patient, defined in the security policy, we divided our testing into two divisions, Pre-Authentication and Post-Authentication, respectively.

We began by using OWASP's list of the top ten most common vulnerabilities and using them as a reference to begin our search.

5 Pre-Authentication

In the following attacks we assume that the adversary has not logged into the service, nor is targeting a logged in user, and explore exploits that we were able to find.

5.1 Cross Site Scripting (XSS)

5.1.1 Overview

XSS is a common web vulnerability that involves injecting JavaScript code into a target web page. This malicious code is introduced through unsanitized user input or even unsanitized URLs, that when properly crafted, cause the website to run the inputted code. Because it is injected into the web page, it cannot be filtered as foreign content or immediately identified as malicious code. Thus, the XSS code runs with all the permissions of the service provider's code and gives an adversary unauthorized power and access to information, depending on the system's design.

The common way to test for XSS vulnerabilities is to verify if an application or web server will react to requests containing scripts with an HTTP response that could be executed by a browser. For example, The Organization's web pages would return an error should any JavaScript tags be passed in. However, as proven by our exploit, this reaction does not automatically guarantee security against XSS. Our group was able to modify a tag in the application by passing additional tag descriptors. Thus without having to pass in a new tag into the application, which would have triggered a response, we were still able to exploit an XSS vulnerability.

XSS vulnerabilities are found by carefully identifying how the website handles user input and any JavaScript passed in the URL of the site. By playing around with these inputs, and seeing how the application reacts, an adversary can determine if the web page is at risk for XSS attacks.

5.1.2 Our Exploit

On The Organization's Contact Us page, they passed in a parameter on the URL which our group used to exploit their site. Originally, the URL was the follow-

ing: `https://[The Organization]/public/forms/Contact.aspx?ctype=2`.

Our initial approach was to pass in a script tag in the URL, which returned a server error. However, after noticing that the value from the parameter `ctype` was being assimilated into the website without any sanitation, our group drafted the following script and replaced the value of `ctype` in the URL to `" accesskey="X" onclick="alert('XSS!')"` like so:

`https://[The Organization URL]/public/forms/Contact.aspx?ctype=" accesskey="X" onclick="alert('XSS!')"`.

This input terminated the value for the parameter `ctype`, and added accesskey and onclick descriptors to the input tag. The changes this payload made on the web page's source code and the effects of pressing the accesskey on the keyboard are illustrated on Figure 1.

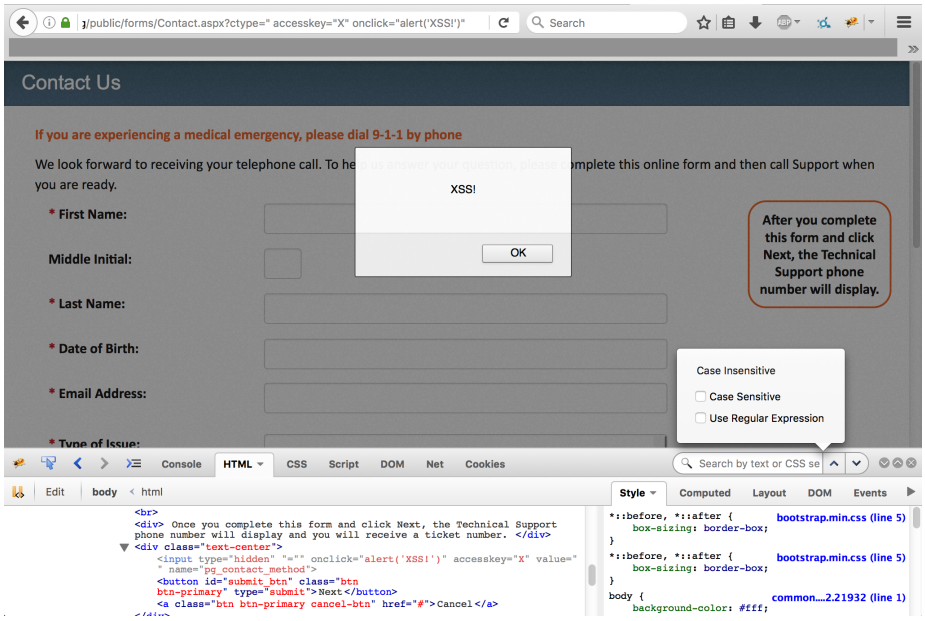

Figure 1: With the additional tag descriptors, the website was able to execute `alert(XSS)`, which could be replaced with malicious code granting an adversary with unauthorized access.

5.1.3 Common Defenses

Common defenses against XSS attacks are to encode every user input before placing it on the source code. Depending on where the user input is going to be placed, this encoding could be html encoding, url encoding, attribute encoding,

CSS encoding, JavaScript unicode encoding, among others. This prevents an adversary from validating their inputs as JavaScript code.

5.2 Buffer Overflows

5.2.1 Overview

Buffer overflows occur when the data written to a buffer continues writing past the allocated memory for a buffer into adjacent memory locations. Because this can lead to an overwrite of the return address on the stack, an adversary can have the return address point to their crafted payload. Although multiple security measures have been implemented to avoid this attack vector, such as a stack canary, making the stack only writable XOR executable (W$\oplus$X), and Address Space Layout Randomization (ASLR), buffer overflows still present a significant vulnerability in the system. This attack vector allow adversaries to run code on the server with administrator permissions.

5.2.2 Our Exploits

Similarly to finding XSS vulnerabilities, Buffer Overflows are found by carefully identifying how the website reacts to large user inputs. Some of the inputs on the login page allowed us to insert an excessive number of characters. If the number of characters and the variable was greater than 2048 characters, then we received a different error response from the server than the typical one. We notified The Organization to have this explored further on the server side, given that our group was not certain what programming language they were using. We found this in these two locations:

```
https://[The Organization URL]/public/forms/faq_pg.html?[2048 cha
racter buffer]
```

```
https://[The Organization URL]/public/forms/Contact.aspx?[2048 ch
aracter buffer]
```

Writing more than the 2048 allowed characters would return the error displayed on Figure 2, shown below.

5.2.3 Common Defenses

Preventing buffer overflow attacks highly depends on the programming language used to code the system. If it is C or C++, there exists safe functions that take in value and cut off the user input once the number of bytes allotted for the buffer are met, such as fgets(), strncpy(), strncat(), among others. In addition, as mentioned previously, making sure that the stack contains a canary, is W$\oplus$X, and is employing ASLR, also helps mitigate these attacks. Although these techniques do not eliminate buffer overflows altogether, they add more hoops

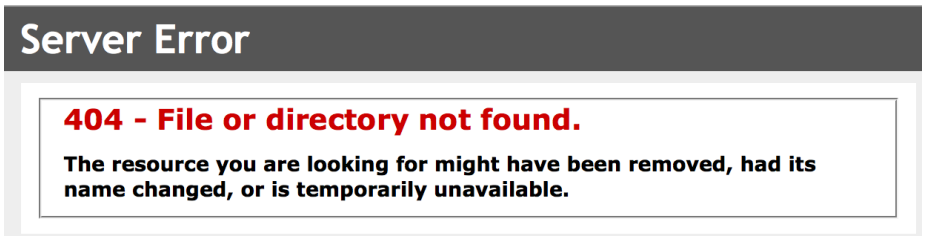

Figure 2: Although this seems like a typical server error, it was different than the other errors encountered attempting to overflow other variable's inputs.

an adversary needs to jump through in order to exploit the vulnerability, thus reducing the likelihood of an attack.

6 Post-Authentication

In the following attacks we assume that some user has logged into the service (not necessarily the attacker) and explore exploits that we were able to find.

6.1 Clickjacking

6.1.1 Overview

Clickjacking involves placing a hidden Iframe behind content that a user would normally view on the application, however clicks through the content up front would also register as valid and authenticated clicks in the Iframe behind. This leads to users unknowingly making changes to their patient portal information without their knowledge, and through clever placement of buttons via CSS by the attacker, the attacker can manipulate exactly what changes the user makes. We were able to successfully place an authenticated user in a hidden Iframe in a users and forward their (valid) clicks through the Iframe.

Clickjacking is limited to mimicking a user's actions from the top level, so it becomes quickly impractical to create such exploits if they require anything more than a few clicks. Also unlike attacks such as XSS, the attacker has no way of viewing the results of their attacks or even confirming that the attacks were successful. Therefore, clickjacking is typically used to abuse services such as Amazon's one-click buy.

6.1.2 Our Exploits

It seemed to us during our analysis of the system that the reason such a vulnerability was left unchecked is because the portal embeds user views and menus into different sites and displays this content to the users via Iframes. This makes it trivial for the attacker to have the user navigate to any page they wish since it is simply a matter of redirecting to the appropriate menu.

Figure 3: Here a user is baited into removing an allergy when trying to view a video.

Some proof of concept attacks that we were able to make include: canceling appointments, removing allergies, requesting prescriptions, and more. In general, any action that only requires a few clicks for confirmation is vulnerable to this type of attack. Below we were able to directly embed the allergy management page into a malicious website.

Clickjacking attacks are particularly dangerous because all input is valid input. There is no way to differentiate unintentional input from what a patient actually wants to do.

6.1.3 Common Defenses

There are a few ways to defend against clickjacking attacks. The most common way is to set the **X-Frame-Options** HTTP header appropriately to prevent foreign domains from embedding the patient portal in their page. This HTTP header allows the service provider to control and specify which domains can set their content within a frame. Browsers would be unable to render the site unless allowed, thereby thwarting any malicious sites like the one we have displayed above.

A more costly and effective solution is to include UI level defensive code that asserts that any frame containing the users content is a top level frame. This ensures that anytime the page is frame it would be visible to the user and they would not be unaware of any of their input.

In terms of the security policy listed above, this clickjacking attack would break the policy stating that only a patient and his doctors are able to update sensitive information.

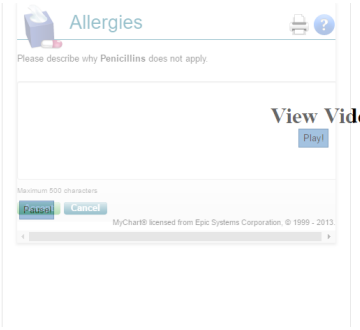

Figure 4: Another example of a user being manipulated via clickjacking.

6.2 Replay Attacks

6.2.1 Overview

A valid user receives his information (such as test results or even viewing their appointments) by logging on to the portal and requesting the information from The Organization's server via something call a packet. This packet would contain much of the information of the request such as the user, the desired information, and typically some sort of authentication keys. The authentication keys are how we can prevent malicious parties from forging these requests. However, the patient portal seemed to have little protection against a malicious party simply repeating valid requests made by a user.

6.2.2 Our Exploits

Figure 5: We were able to replay packets to receive confidential information.

We were able to replay packets sent by simply adding the **X-Forwarded-For:** header to our request. This header tells the server that we are trying to proxy for their client. Replaying these request worked throughout the logged in

user's session. Through this attack, we were able to get HTML for pages that contained sensitive medical information without the client's knowledge.

6.2.3 Common Defenses

Replay attacks can be defended against by using a few different techniques. The most common technique is to require some sort of timestamp or id hashed along with whatever information is being sent by the user. The server is then able to only accept results that are within some reasonable threshold of the original request and can simply ignore those which are on time. This defense does not prevent sufficiently quick attackers from repeating the request within the timing threshold.

7 Conclusions

In our analysis of the patient portal, we were able to find some potentially serious exploits, as well as suggest potential fixes to these vulnerabilities. Although we were also able to find other potential issues (such as a potential session high-jacking vulnerability), we could not reliably reproduce these, so we were not comfortable reviewing them as exploits. We were only able to spend two days working with the system. Therefore, more vulnerabilities potentially exist that we were unable to discover in our limited time. As such, we recommend that any future investigations use our report as a starting point.

8 Permission and Acknowledgements

All tests were performed under the explicit permission of The Organization. We would like to thank the people of The Organization for so generously working with us and allowing us to complete our project.

Based on the agreements we signed with The Organization, this report is not to be released until The Organization agrees to a release. The members of this team will contact the course staff once we have received approval for it to be posted.

We would also like to thank the 6.857 staff and everyone who helped us at The Organization for their support and guidance in this project!

9 References

1. The State of Cybersecurity in Healthcare Organizations in 2016
2. https://www.owasp.org/index.php/Top_10_2013-Top_10
3. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
4. https://www.owasp.org/index.php/Buffer_Overflow

5. https://www.owasp.org/index.php/Clickjacking

6. https://www.owasp.org/index.php/Capture-replay