Type Conversion Errors

Black Hat USA - 2007

June 25, 2007

Presented By:

Jeff Morin
Sr. Security Consultant, SpiderLabs
jmorin@atwcorp.net

1 OVERVIEW

In the realm of application testing, one of the major, but most often overlooked vulnerabilities, is that of type conversion errors. These errors are the result of input variable values being used throughout the many areas and codebases that make up the application. Reused input variable values can be potentially treated as different data types throughout their processing.

The application functions correctly and without issue because the values of the input variable are anticipated, even though they are treated in different areas as different data types. The issue arises then when a value is input into one of these variables that is crafted in such a way as to be successfully manipulated by some data types, while failing others, resulting in the application behaving in unanticipated and potentially dangerous ways.

These vulnerabilities are application logic flaws which are much more difficult to identify than simple input validation errors, such as error-based SQL injection or XSS, as they don't readily display success or failure, rather can manifest themselves in other areas or at a later time. This also makes them very dangerous in that the application behaves in completely unanticipated ways, potentially resulting in circumvented authentication and authorization, Denial of Service, elevated privileges, etc.

2 THE PROBLEM

Origin of the Vulnerability

Developers are aware of the different data types that a variable can be assigned, and how each data type is interpreted. Certain data types are able to take in multiple inputs of varying construction, but which ultimately evaluate to the same value.

An example of this is a variable set to a data type of integer. The following values set to a variable of data type **int** would be evaluated as the same number.

01 == 1
00000000002 == 2

The same goes for a variable set to a data type of **float**:

1.00000 == 1
000000.2 == 0.2
00000.30000 == 0.3

The data type of a string on the other hand does not allow for different inputs to evaluate to the same value:

01 <> 1
1.00000 <> 1

Thus, it is clearly possible to assign different inputs to a variable of data type **int** or **float** and have them be evaluated as the same value, whereas with strings this is not possible. Herein lies the origin of the type conversion vulnerability.

The Vulnerability

It has already been illustrated how variables of certain data types can be assigned multiple inputs which effectively evaluate to the same value, and with other data types, this is not possible. The problem then stems from the fact that data assigned to a variable of type int or float, when compared to another variable of the same data type, will evaluate correctly in a conditional statement. In contrast, the same data used in a conditional statement utilizing a string data type for comparison may evaluate incorrectly or ultimately fail when compared to the original int or float variable.

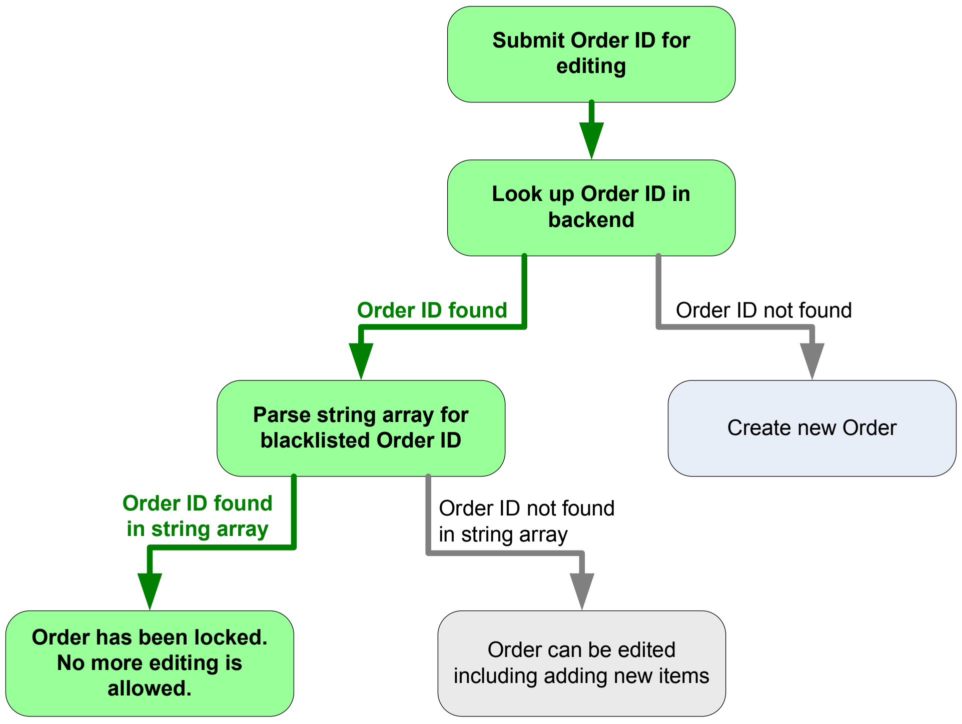
The results of failing conditional statements in other areas of the application are not immediately apparent, as error messages do not usually occur, rather the application behaves in an unanticipated manner. Depending on the application and function in which the conditional is evaluating, this can result in bypassed authorization or a Denial of Service (DoS) condition. If the data is stored in a database, these functions and conditionals may not be executed until a much later time, which again makes identification of the vulnerability a great deal more difficult.

An Example

The following is an example of how an application which processed data in some areas as an integer and another as a string resulted in a major vulnerability.

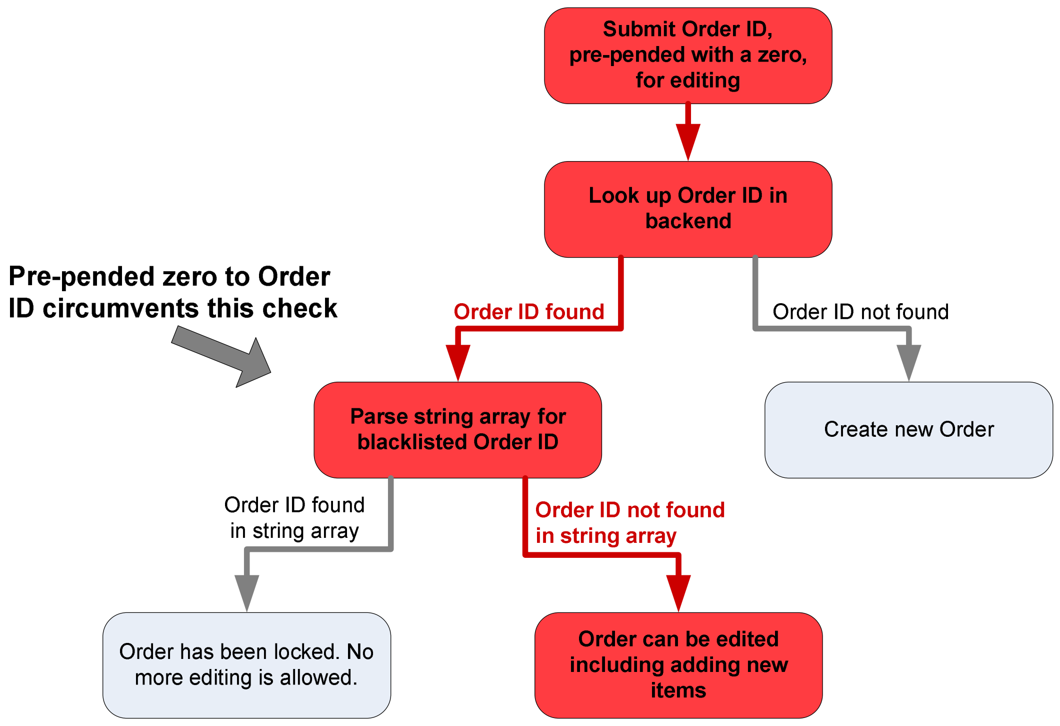
The application was a typical web based ecommerce site which utilized a shopping cart to browse the site and select items for purchase. The cart could be edited at any time prior to actually purchasing the items and was tracked with an order ID number. A user could stop shopping and come back later and continue where they left off. Once the items had been purchased though, the order was locked in and the user could not modify the contents as that would have resulted in the user getting items for free. This was accomplished by taking the Order ID and placing it into a string array of blacklisted IDs. If a user went back to check on the order items, the Order ID was first referenced in the database for the order items, and then the array was parsed to determine if the ID was blacklisted, and thus not allowed to be edited.

Process flow for attempting to edit a locked order:


Submit Order ID for editing
Look up Order ID in backend
Order ID found
Order ID not found
Parse string array for blacklisted Order ID
Create new Order
Order ID found in string array
Order ID not found in string array
Order has been locked. No more editing is allowed.
Order can be edited including adding new items

The problem resulted from the Order ID being tracked and verified normally as a variable of data type int, but then used in conditional statement against a string array. By pre-pending a zero to the beginning of the Order ID and submitting it to be edited, an attacker was able to successfully lookup the targeted Order ID in the database, and subsequently bypass the string array, because the Order ID, pre-pended by a zero, did not match any of the blacklist entries in the string array. This allowed the attacker to pay for an order, and then go back and add additional items to their order for free.

Process flow for attempting to edit a locked order with modified Order ID:


Submit Order ID, pre-pended with a zero, for editing
Look up Order ID in backend
Pre-pended zero to Order ID circumvents this check
Order ID found
Order ID not found
Parse string array for blacklisted Order ID
Create new Order
Order ID found in string array
Order ID not found in string array
Order has been locked. No more editing is allowed.
Order can be edited including adding new items

This example illustrates the danger of processing data as different data types throughout an application.

3 CONCLUSION

Type conversion errors are a common problem in web applications which are typically overlooked while performing security testing or code reviews. The disconnect between multiple developers utilizing data which is processed in multiple areas as different data types can result in unanticipated application functionality, such as bypassing of application access restrictions. Developers and security testers must all be aware of how type conversion errors occur, the potential risk to the application and data contained therein, and the need for additional focused application logic testing during development and prior to going live.