

## Teaching Case

# A Framework for Cybersecurity Educational Activities

Br. David Carlson  
david.carlson@stvincent.edu  
CIS Department, St. Vincent College  
Latrobe, PA 15650

Anthony Serapiglia  
Anthony.Serapiglia@stvincent.edu  
CIS Department, St. Vincent College  
Latrobe, PA 15650

### Abstract

A framework is presented that uses web pages, stand-alone programs, and web applications to teach cybersecurity principles related to software development problems such as buffer overflows or a lack of filtering of user input in a web application. Each of the web pages starts with these headings: Software Development, What Could Possibly Go Wrong? This describes well the focus of the exercises. The particular examples used assume little prior knowledge and supply all that is needed to solve the exercises. In this form, they have been used with prospective students, visitors, young teens at summer camp, and students in introductory programming classes such as CS 2. The materials can be modified to be more demanding for college students who have sufficient background to produce the solutions themselves.

**Keywords:** framework, cybersecurity, software development, software errors, secure coding

### 1. INTRODUCTION

Cybersecurity studies encompass a wide spectrum of material. A thorough program should include material that also covers multiple different avenues of instruction, from lecture and research to practical labs and exercises. Many topics within the more technical areas of cybersecurity are often difficult for beginning students to imagine or visualize until they experience them firsthand. Unfortunately for many programs, limited resources of time/money/equipment and exact skill set expertise can be barriers to providing these experiences in the classroom.

The purpose of this paper is to provide multiple exercises that allow easy adoption into a

classroom to display several fundamental cybersecurity topics with student interaction. The exercises fit a common framework so that some or all of them can be used together, but they can be used in a stand-alone fashion as well.

#### Prerequisites

The examples currently in the framework do not require a lot of technical background. However, it is assumed that the students can navigate a collection of web pages, are familiar with running small stand-alone programs as well as web applications and understand the concepts of a function and the graph of a function.

## 2. EXERCISES

The series of examples in this framework can be accessed in a browser using the URL for the series of online web pages (<https://cis.stvincent.edu/carlson/cs205/exclude/New/softdev.html>). These web pages all have the same layout so as to make the directions and explanations easier to follow. The very first web page provides some background and initial directions for accessing four stand-alone executables, the first four software examples. The last two examples are web applications. A brief conclusion is given at the end of the last example but could be moved to a separate page, if desired.

### Software Development

Most commercial software works well most of the time. Software that students write, sometimes works and sometimes does not. Even commercial software sometimes has problems. Software can even fail to work "right" in various ways. Some of these failures have serious consequences. Some of them can allow attackers to compromise our computer systems. The goal of these exercises is to allow you to try out, in a safe setting, examples where software does not behave as it should.

#### Example 1

In the first exercise we investigate what is called an ill-conditioned problem. In this type of problem, small changes to the input can result in huge changes in the output. For example, if you measure the length of an object twice, you might well get slightly different lengths. If the computation done with that length is ill-conditioned, you might get wildly different answers for those two lengths. You are left to wonder if either answer is right or even if there is a good answer.

#### Example 2

Integer overflow is illustrated in this example. It is well known that there is a largest positive integer with the usual type of integer available in computer software. Less well known is the fact that if you reach this largest positive integer and add 1 to it, you reach the smallest negative integer. Note that computer software often processes data where some variable successively takes on each and every number value from a large range of values. Normally this works well, but there are possible hazards, such as trying a range that goes beyond the largest positive integer.

#### Example 3

Stack overflow is another well-known problem. Each time a computer program starts another

function, a small region of memory is used to record information about that function. However, if that function needs to use a second function, another region of memory is needed. If that second function needs to use a third function, another memory region is used, etc. In some cases, there can be a long sequence of functions used (perhaps the same function used multiple times), which can result in many chunks of memory being used in an area of memory called the stack. If the stack does not have enough space to hold those many chunks of memory, we get stack overflow. The usual result is that the program fails. It crashes, quits working, which is not a nice thing for a program to do,

#### Example 4

This example shows another type of overflow: buffer overflow. A buffer is a region of memory used to hold data items. In this example, the data is a string that you are prompted to enter when you run the program. If this data gets copied into another string, and that string isn't long enough to hold the data, that is a buffer overflow. You might think that any extra letters that do not fit would be thrown away. It depends on the coding. In this case, the extra letters overwrite other things that are stored in memory. This corruption of data, and other items in memory, can lead to erratic program behavior. It can even lead to the program crashing or a serious security breach!

#### Example 5

We now switch to web applications.

This example shows you an SQL injection attack against a pretend banking site. Many web applications use a database to hold data about products, accounts, purchases, etc. SQL is a fairly standard language used to insert and retrieve data from a database. Sometimes a flaw in a web application allows a user to enter part of an SQL instruction as data, data that gets used as one component of the web app's SQL commands to the database. This can allow a malicious user to see data that belongs to other users and maybe even to change or delete data that belongs to other users.

#### Example 6

In this last example, we have a cross-site scripting attack (XSS for short). One way this can happen is when users are allowed to post messages to a web site and other users are allowed to read those messages. A malicious person might post something that is JavaScript code instead of a real message. When other users visit this site, which they trust, the malicious person's code runs. This is code from a different

source, a different site, but the other users may not realize that the site that they trust contains something from outside that maybe they should not trust.

### 3. QUESTIONS

What might go wrong if a user types in more data as input into an application than will fit in an array designated to hold this data?

Is there any limit to the number of function calls that an application can have active at once?

What happens if a program starts at 1, adds 1, adds 1 again, and so forth? Can this go on indefinitely or does something else happen?

If a computation that normally works well fails to work at all for one or more inputs, what might go wrong for inputs very close to where the computation fails?

If a web application asks for user input in the form of text that can be seen by other people using the same web application, what might be entered that could be malicious and what defenses might be used to block such an attack?

If a web application takes user input and incorporates it as part of an SQL command to extract data from a backend database, what might an attacker enter as input in order to try to get the application to reveal more of its data than it should?

### 4. CONCLUSIONS

A series of examples has been presented that can be used to teach students and other interested persons about what can go wrong in software development as well as some secure coding principles, such as the importance of filtering user input, that can be used to fix these problems. The particular online version that is shown in Appendix 1 gives rather complete directions so as  
NJ: Prentice Hall PTR.

to be accessible to people with only a few simple prerequisites. The code for this series of examples is available for download (<https://cis.stvincent.edu/Security.tar.gz>). Small adjustments, such as in the links from one page to the next, would be needed. Instructors who have students with a good background in cybersecurity and software development may well wish to remove many of the hints on how to solve these problems so as to make them more challenging.

The overall framework of a series of web pages containing instructions and hands-on activities to reinforce learning is a helpful one. It might be used with many topics (in cybersecurity and in other areas) and in many settings where the instructor wishes to utilize online learning reinforced with hands-on activities.

### 5. REFERENCES

- Carlson, D., "Teaching Computer Security," *SIGCSE Bulletin*, vol. 36, no. 2, pp. 64-67, June 2004.
- Carlson, D. "Software Development: What Could Possibly Go Wrong?" <https://cis.stvincent.edu/cyber/softdev.html>
- Chapra, S. C., & Canale, R. P. (2006). *Numerical methods for engineers* (Fifth ed.). Boston: McGraw-Hill Higher Education.
- Conklin, W. A. (2010). *Principles of computer security: CompTIA security and beyond* (Second ed.). New York: McGraw-Hill.
- Du, W. (2017, October). SEED Project: Hands-on labs for security education. Retrieved August 11, 2019, from <https://seedsecuritylabs.org/> NSF-funded project.
- Seacord, R. C. (2008). *Secure coding in C and C++* (SEI). Upper Saddle River (NJ): Addison-Wesley.
- Skoudis, E., & Zeltser, L. (2008). *Malware: fighting malicious code*. Upper Saddle River,

# Appendix 1

Included in Appendix 1 is a collection of the web pages utilized in the exercises. To maintain the anonymity of the peer review process, links have been disabled that would show institution and author name in the URL. These links and downloads will be made available to the public should this paper be accepted into the proceedings.

## Software Development

### What Could Possibly Go Wrong?

#### Background

- Most commercial software works well most of the time.
- Software that students write sometimes works and sometimes not.
- Even commercial software sometimes has problems.
- Software can fail to work right in various ways.
- Some of these failures have serious consequences.
- Some of them can even allow attackers to compromise our computer systems.
- The goal of these exercises is to allow you to try out, in a safe setting, examples where software does not behave as it should.

#### Directions

- If your instructor has supplied the 4 demo programs on your computer, do this section:
  1. In Windows, start File Explorer (or its equivalent).
  2. Your instructor will then tell you to go to a certain location in File Explorer.
  3. Double click on the folder SecurityDemoSoftware.
  4. You should then see 4 files. These will be used in the examples described next.
- If you instructor tells you to download the demo programs, do this section:
  1. Click on [SecurityDemoSoftware Tar File](#).
  2. Your browser should then allow you to download this zip file of software.
  3. Choose to save that zip file.
  4. Use File Explorer to open your Downloads folder.
  5. Right click on the file SecurityDemoSoftware.zip that you find there.
  6. Select the Extract All option and tell it to extract to your Documents folder.
  7. Navigate to that Documents folder.
  8. You should have there a folder named SecurityDemoSoftware. Double click it to reveal the software that is inside.

#### Next

Follow [this link to try out the first example](#).

Last updated: May 25, 2019

# Software Development

## What Could Possibly Go Wrong?

### Example 1

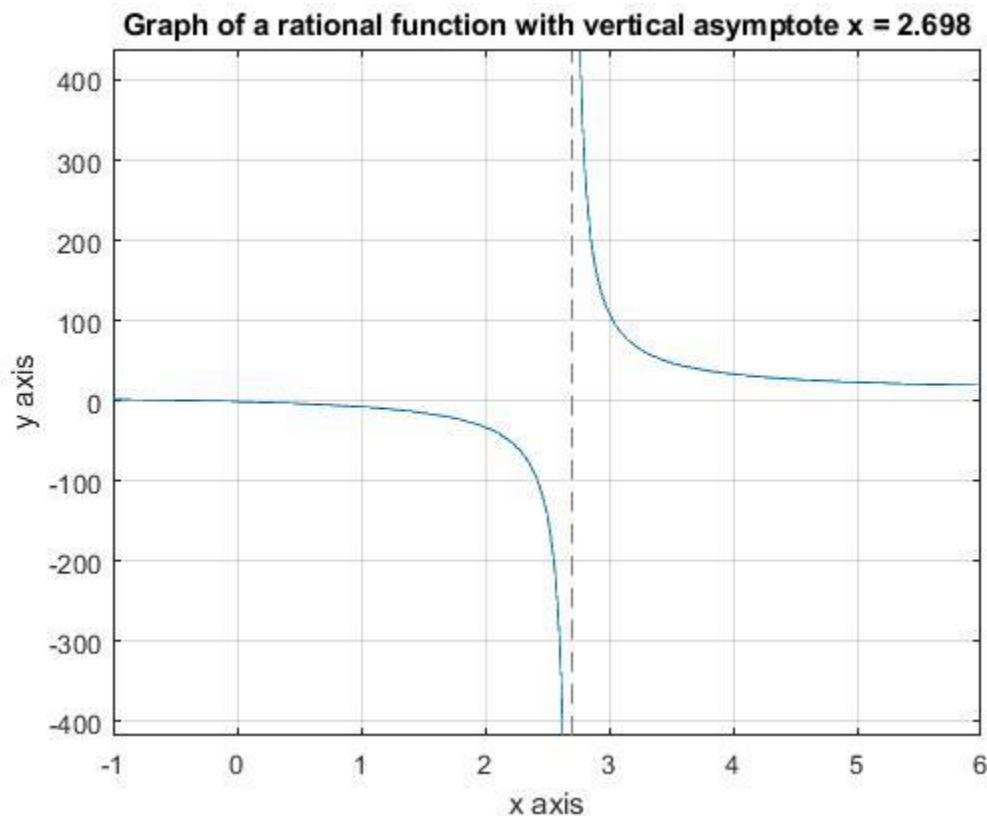
#### Background

- Here we investigate what is called an ill-conditioned problem.
- In this type of problem, small changes to the input can result in huge changes in the output.
- For example, if you measure the length of an object twice, you might well get slightly different lengths.
- If the computation done with that length is ill-conditioned, you might get wildly different answers for those two lengths.
- You are left to wonder if either answer is right or even if there is a good answer.

#### Directions

1. In your SecurityDemoSoftware folder, double click the program named IllConditionedProblem.
2. You may get a popup that says, "Windows protected your PC".
3. If so, click on the More info link and then click the Run anyway button.
4. The program should start and display 3 values of a function named g.
5. Then the program asks you to enter two x values for the function. Do so. For example, you might type 2.697 for one of them.
6. Then try a number near 2.699 to see what output you get.
7. Note that you get wildly different output values.
8. Any small change (error) in the input value x (at least for x values near 2.698) is thus likely to be magnified into a large change (error) in the output for g(x).
9. Press Enter one final time to end the program.
10. You can run the program again if you wish to compute more function values.

## Explanation



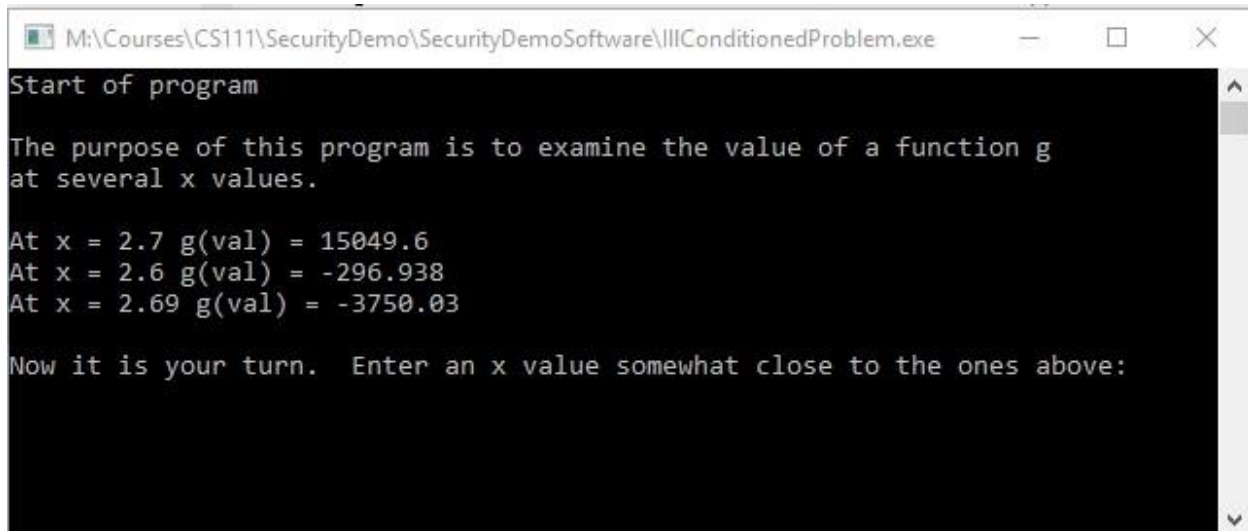
- The function  $g$ , as seen in this graph, is undefined at 2.698 and varies a lot for  $x$  values near that.
- In contrast, for  $x$  values not close to 2.698, the function is well-behaved.
- Notice how the graph shoots up to plus infinity on one side of 2.698 and to minus infinity on the other side.
- The formula for the function, an example of a rational function, reveals why it behaves this way:  $g(x) = (10x + 3.1) / (x - 2.698)$ , so that it tries to divide by zero when  $x$  is 2.698.
- The take-away is that ill-conditioned problems might give us rather inaccurate answers. Thus, something can indeed go wrong here.

## Next

Follow [this link to try out the next example.](#)

Last updated: May 24, 2019

Image inserted below to show what students see when they run the IllConditionedProblem.exe program for example 1 above:



```
M:\Courses\CS111\SecurityDemo\SecurityDemoSoftware\IllConditionedProblem.exe
Start of program
The purpose of this program is to examine the value of a function g
at several x values.
At x = 2.7 g(val) = 15049.6
At x = 2.6 g(val) = -296.938
At x = 2.69 g(val) = -3750.03
Now it is your turn. Enter an x value somewhat close to the ones above:
```

## Software Development

### What Could Possibly Go Wrong?

#### Example 2

##### Background

- Integer overflow is illustrated in this example.
- It is well known that there is a largest positive integer with the usual type of integer available in computer software.
- Less well known is the fact that if you reach this largest positive integer and add 1 to it, you reach the smallest negative integer.
- Note that computer software often processes data where some variable successively takes on each and every number value from a large range of values. Normally this works well, but there are possible hazards.

##### Directions

1. In your SecurityDemoSoftware folder, double click the program named IntegerOverflow.
2. The program should ask you to enter a starting number.
3. Enter 2 billion, which is 2,000,000,000 but enter it without the commas. They are shown here so that you can see how many zeros to use.
4. The program will then ask you for an increment that it will repeatedly add on.
5. Enter a number in the range from 10 million to 20 million. For example, to use 10 million, enter 10,000,000 without the commas.
6. Press Enter repeatedly until you have gone beyond the largest positive integer and are into the negatives.
7. If you continue to press Enter until you have exhausted the negatives, what happens?
8. Continue to press Enter until you understand the pattern.
9. Use CTRL c to end the program. That is, hold down CTRL and press c.

## Explanation

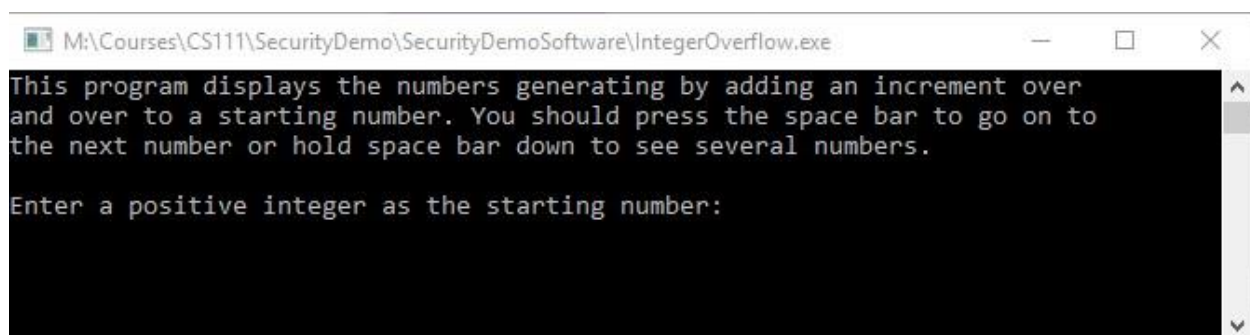
- For ordinary integers on a computer, the number system wraps around in a big circle.
- This means that if we try to calculate with numbers that might get too large, we may get an answer that makes no sense, such as a negative answer when a positive one is expected.
- Truly something has gone wrong: integer overflow!

## Next

Follow [this link to try out the next example](#).

Last updated: May 23, 2019

Image inserted below to show what students see when they run the IntegerOverflow.exe program for example 2 above:



```
M:\Courses\CS111\SecurityDemo\SecurityDemoSoftware\IntegerOverflow.exe
This program displays the numbers generating by adding an increment over
and over to a starting number. You should press the space bar to go on to
the next number or hold space bar down to see several numbers.

Enter a positive integer as the starting number:
```



# Software Development

## What Could Possibly Go Wrong?

### Example 3

#### Background

- Stack overflow is another well-known problem.
- Each time a computer program starts another function, a small region of memory is used to record information about that function.
- However, if that function needs to use a second function, another region of memory is needed.
- If that second function needs to use a third function, more memory is used, etc.
- In some cases, there can be a long sequence of functions used (perhaps the same function used multiple times), which can result in many chunks of memory being used in an area of memory called the stack.
- If the stack does not have enough space to hold those many chunks of memory, we get stack overflow.
- The usual result is that the program fails. It crashes, quits working.

#### Directions

1. In your SecurityDemoSoftware folder, double click the program named StackOverflow.
2. The program then outputs a long sequence of numbers: -1, -2, -3, ..., one number each time a function is used and needs another chunk of memory in the stack.
3. When the program runs out of stack space, a window will pop up and tell you that StackOverflow.exe has stopped working.
4. If the window has a link labeled More Info, you can click it and see what it tells you.
5. Click the button to close the program. If there is a Debug button, do NOT use it as it will cause you a lot of delay.

#### Explanation

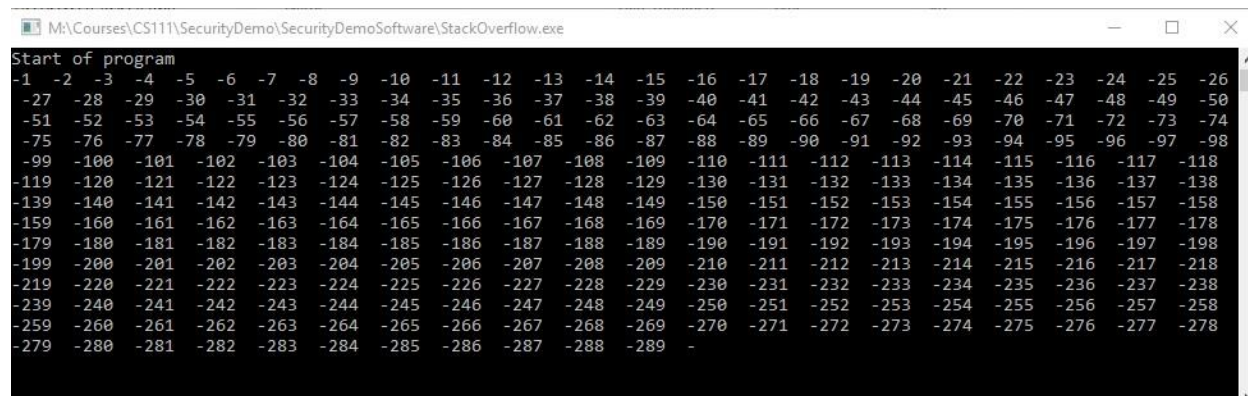
- In our case, the stack overflow did not cause any large amount of harm.
- However, it did keep us from using our software.
- If this happened when we were using professional software to work on something mission-critical, it could be very important. In such a case, something would be very wrong.

#### Next

Follow [this link to try out the next example.](#)

Last updated: May 23, 2019

Image inserted below to show what students see when they run the IllConditioned.exe program for example 3 above:



```
M:\Courses\CS111\SecurityDemo\SecurityDemoSoftware\StackOverflow.exe
Start of program
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26
-27 -28 -29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43 -44 -45 -46 -47 -48 -49 -50
-51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -64 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74
-75 -76 -77 -78 -79 -80 -81 -82 -83 -84 -85 -86 -87 -88 -89 -90 -91 -92 -93 -94 -95 -96 -97 -98
-99 -100 -101 -102 -103 -104 -105 -106 -107 -108 -109 -110 -111 -112 -113 -114 -115 -116 -117 -118
-119 -120 -121 -122 -123 -124 -125 -126 -127 -128 -129 -130 -131 -132 -133 -134 -135 -136 -137 -138
-139 -140 -141 -142 -143 -144 -145 -146 -147 -148 -149 -150 -151 -152 -153 -154 -155 -156 -157 -158
-159 -160 -161 -162 -163 -164 -165 -166 -167 -168 -169 -170 -171 -172 -173 -174 -175 -176 -177 -178
-179 -180 -181 -182 -183 -184 -185 -186 -187 -188 -189 -190 -191 -192 -193 -194 -195 -196 -197 -198
-199 -200 -201 -202 -203 -204 -205 -206 -207 -208 -209 -210 -211 -212 -213 -214 -215 -216 -217 -218
-219 -220 -221 -222 -223 -224 -225 -226 -227 -228 -229 -230 -231 -232 -233 -234 -235 -236 -237 -238
-239 -240 -241 -242 -243 -244 -245 -246 -247 -248 -249 -250 -251 -252 -253 -254 -255 -256 -257 -258
-259 -260 -261 -262 -263 -264 -265 -266 -267 -268 -269 -270 -271 -272 -273 -274 -275 -276 -277 -278
-279 -280 -281 -282 -283 -284 -285 -286 -287 -288 -289 -
```

# Software Development

## What Could Possibly Go Wrong?

### Example 4

#### Background

- This example shows another type of overflow: buffer overflow.
- A buffer is a region of memory used to hold data items.
- In this example, the data is a string that you are prompted to enter when you run the program.
- If this data gets copied into another string, and that string isn't long enough to hold the data, that is a buffer overflow.
- You might think that any extra letters that do not fit would be thrown away.
- Instead, the extra letters overwrite other things that are stored in memory.
- This corruption of data, and other items in memory, can lead to erratic program behavior.
- It can even lead to the program crashing or a serious security breach!

#### Directions

1. In your SecurityDemoSoftware folder, double click the program named BufferOverflow.

2. The program asks you to enter a message. Type your full name and press Enter.
3. The program was designed so that it can read in a message up to 64 characters long.
4. However, internally it then copies your message into a string that can only handle 8 characters, thus potentially causing a buffer overflow.
5. Your name is probably not long enough to cause any noticeable problem.
6. Often the data has to overflow the string by quite a bit before a problem is seen.
7. On a second attempt in running this program, enter a string of 60 characters. You can do this by typing 0123456789 six times with no spaces. Press Enter and see what you get.
8. Typically, the result will be a program crash.
9. If the popup window has a More Info link, you can click on that to see what it might tell you.
10. Click on the button to end this faulty program.

## Explanation

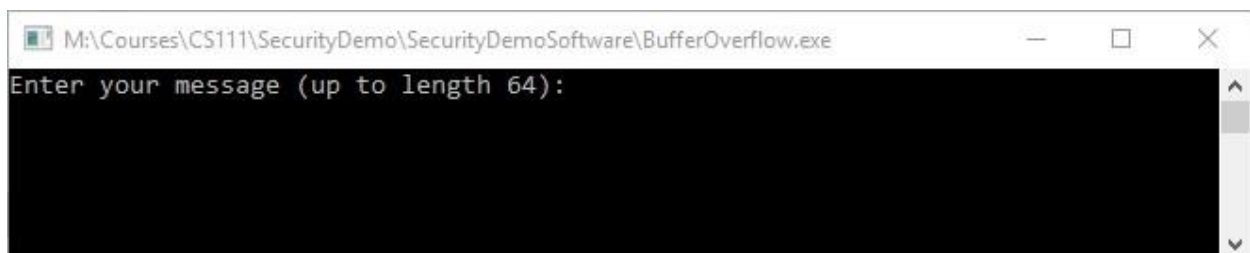
- This buffer overflow overwrites something important in the functioning of the program.
- This is probably the information about where the program should go to after processing the function that copied your long string into the string that holds only 8 characters.
- At this point, the program cannot correctly get out of that function and return to where it should be for the next part of the program. Thus, the program typically crashes.
- How serious can this be? Buffer overflows have often allowed bad actors working over the Internet to cause web applications to crash in such a way that they, the bad guys, are given complete access to the particular web server. From there, they have access to go on to cause additional (and possibly severe) problems.

## Next

Follow [this link to try out the next example.](#)

Last updated: May 23, 2019

Image inserted below to show what students see when they run the BufferOverflow.exe program for example 4 above:



# Software Development

## What Could Possibly Go Wrong?

### Example 5

#### Background

- We now switch to web applications.
- This example shows you an SQL injection attack against a pretend banking site.
- Many web applications use a database to hold data about products, accounts, purchases, etc.
- SQL is a fairly standard language used to insert and retrieve data from a database.
- Sometimes a flaw in a web application allows a user to enter part of an SQL instruction as data, data that gets used as one component of the web app's SQL commands to the database.
- This can allow a malicious user to see data that belongs to other users and maybe even to change or delete data that belongs to other users.

#### Directions

1. Open a new tab in your browser or use a different browser for the following steps.
2. Copy the following into the browser's address bar:  
<https://cis.stvincent.edu/cyber/BankSQLInjection/bank.html>
3. You will see a form where you can fill in your pretend user ID and password to log into a pretend online banking site.
4. Fill in the number 5 as your user ID and 4rT=7abt as your password.
5. Then click the Submit button.
6. This should work normally, showing you your bank account information, including the credit card number for a credit card issued to you by your bank, your current available credit on that card, your saving account balance, and your checking account balance.
7. Use your browser's back button to go back to the login page for this pretend bank.
8. This time, use the same user ID of 5, but change the password to: 4rT=7abt" OR "1  
Note that the last character is the digit 1.
9. To accurately enter this supposed password, you might want to copy and paste it.
10. This time when you click the submit button, you see the bank account information for all the bank customers, a total of 5 customers.
11. You are seeing the private information for all the users, including their credit card numbers!
12. See the image below that shows the table of information for all users that is produced by carrying out this SQL injection attack.

**Online Banking**

To access your account, fill in your data and click on Submit.

Your user id:

Your password:

Version 1.1

**Online Banking**

Bank Customer: 5

User ID	First Name	Last Name	Credit Card Number	Card Type	Credit Available	Savings Account Balance	Checking Account Balance
1	Sally	Wilson	1111222233334444	Master Card	2567.31	39251.90	6386.24
2	Bob	Jamison	1234123412341234	Visa	574.62	10583.76	4931.28
3	Al	Lytle	4444222211115555	American Express	866.37	25935.57	7034.68
4	Tan	Nguyen	7423806526541284	Master Card	1206.00	8438.61	2539.84
5	Lisa	Goldberg	3857129378049371	Visa	3578.26	46822.75	9628.53

Version 1.1

### Explanation

- Once the particular user ID of 5 and the normal password are filled in, this web app uses the following SQL command or a very similar one: `SELECT * FROM Users WHERE userid = "5" AND password = "4rT=7abt";`
- But by entering a malicious password of `4rT=7abt" OR "1` the SQL command becomes: `SELECT * FROM Users WHERE userid = "5" AND password = "4rT=7abt" OR "1";`
- The 1 is taken to mean true, and doing an AND takes precedence over doing an OR.

- Thus, the condition `userid = "5" AND password = "4rT=7abt"` is irrelevant. Since the 1 means true the entire condition is always true.
- Thus, the data for all user IDs in the Users table is returned by this command. The attacker has found a way to use your web app to steal private data.
- How can this type of attack be prevented? What is needed is good filtering of the user input, filtering that rejects input that contains items that do not belong in a password for this system. Our particular malicious password could be rejected because it contains double quotes and spaces. Thus, we might restrict the passwords to strings that contain only letters and digits, in order to prevent the above type of attack.

## Next

Follow [this link to try out the next example](#).

Last updated: June 18, 2019

# Software Development

## What Could Possibly Go Wrong?

### Example 6

#### Background

- In this last example, we have a cross-site scripting attack (XSS for short).
- One way this can happen is when users are allowed to post messages to a web site and other users are allowed to read these messages.
- A malicious person might post something that is JavaScript code instead of a real message.
- When other users visit this site, which they trust, the malicious person's code runs. This is code from a different source, a different site, but the other users may not realize that the site that they trust contains something from outside that maybe they should not trust.

#### Directions

1. Modern browsers provide some protection against attacks. To get the following harmless attacks to work, you may have to try a couple of different browsers and/or try the attack again with some of the protections turned off.
2. First, you need a unique 2-digit number from 01 to 32. Get the number to use from the person conducting this demo or if doing these in a group by yourselves, decide among



yourselves who gets what number. This example does not work well if more than one person has the same number.

3. In a new tab in your browser (or in a different browser), copy the following into the browser's address bar, where you carefully replace the NN by your particular 2-digit number: `https://*****/storeNN/Comments.php`
4. Note that a number such as 3 should have the digit 0 in front of it. Thus 03 would be used to replace the NN, etc.
5. You should be given a simple page, part of an online sales site, describing a type of mug and asking for customer comments about it.
6. In the comment box, write a safe comment such as `Nice Mug`.
7. Click the Submit Your Review button. You should see your comment now posted on the page.
8. If you (or anyone else) look at this particular mug page, the comment you posted will be seen.
9. Next, type the code shown in the small image below (or in the image to the upper right) into your comment box. An image is used here so that the popup does not run when you simply view this page of directions. Note: It may be faster to copy and paste this text `alert("You've been hacked!");` and then put the script tags around it in your comment box.

```
<script>alert("You've been hacked!");</script>
```

10. Click the Submit Your Review button.
11. You should either get a popup box that says `You've been hacked!` or a warning that something was blocked.
12. If the browser blocked things, just click your browser's button for reloading the page (or you can go to the end of the address in the address bar and press Enter). You may then be able to get the popup. If not, see below on how to turn off the blocking of popups.
13. Click the reset button on your mug page to put things back to normal. This will clear out the code for the popup and any other messages that you stored on this page.
14. As a last cross-site scripting attempt, type the code shown below into your comment box. Once again, it is probably faster to copy and paste this text `window.location="http://www.google.com";` and then put the script tags around it in your comment box.

```
<script>window.location="http://www.google.com";</script>
```

15. Click the Submit Your Review button, of course.
16. Now whenever you or anyone else views your mug page, the browser will be redirected to Google's web site.
17. Use your browser's back button to get back to your mug page, if possible. Then click the reset button on your page to put things back to normal.
18. If this XSS attack does not work in one browser, try another browser.
19. If you are using the Chrome browser and the above attacks were blocked, one way to get them to work is as follows: Open Settings, go to Advanced Settings, Privacy and security, Site Settings, Popups and redirects. Change the first setting from Blocked to Allowed. After you have run your harmless XSS attacks, put that Chrome setting back to Blocked, as that is much safer.


Add your review of this product:

```
<script>alert("You've been hacked!");</script>
```

Submit Your Review

### The Perfect Mug

By Perfection, Inc.



\$12.99  
4 for \$50.00  
Free shipping on orders of 4 or more  
In stock

Features:  
Unbreakable  
Insulated. Will not burn your hand.  
Customized engraving added to your request

Comments from customers who have purchased this product:  
Great mug! I use it all the time.

You've been hacked!

OK

## Explanation

- The injected code runs for every user who visits your mug page, including yourself.
- Getting redirected to some site like google.com might be annoying to users, but think what an attacker might do. Such a person might redirect users to a server that attempts to download malware to any computers that visit that server. This is called a "drive-by download" attack.



- How can cross-site scripting attacks be prevented? Basically, by the same method as in the last example: Use good filtering of the user input, filtering that rejects input that contains any kind of code, html tags, etc.

## Final Items

- If you have the time, watch one of the following videos of the Ariane 5 rocket explosion:
  - [Long \(4 min\) video: Ariane 5 rocket explosion.](#)
  - [Short video: Ariane 5 rocket explosion.](#)
- If neither of those links works, try searching YouTube for "Ariane 5 explosion".
- This explosion was due to software that tried to store the horizontal velocity of the rocket, a rather long number, into a short integer. It did not fit because the horizontal velocity of this version of the rocket could be larger than in the previous version.
- This resulted in an invalid number for that short integer. Unfortunately, that bad integer was used to control the rocket and sent it so far off course that the rocket broke up.
- This is similar to some of the overflow problems that the earlier examples in this demo showed you.
- Hopefully you learned a few things from these examples, things that will help you as a user of software and things that will help you to write safer software if you plan to become a software developer.

Last updated: June 18, 2019