# Insidious Inputs – Three Types of Common Software Vulnerabilities


| | |
|---|---|
| **Author** | josh |
| **Date** | 2025-01-05 |


# Overview

Software runs the digital world, and while computers will faithfully execute every instruction given to them, sometimes those instructions are poorly written. While software engineers make every attempt to write clean, well-tested, and secure code, security holes often arise in their code. There's a myriad of ways attackers can exploit vulnerable software to their ends. In this article, we'll focus on three common input-related vulnerabilities that allow attackers to achieve some different, malicious objectives. One common theme among these will be *input*, the instructions given by the users of the applications to execute some tasks. As we'll see, input can be downright dangerous to software and information systems if not handled properly. Let's learn some fun ways to exploit poorly handled input, and some ways to mitigate those attacks.

# Input Gone Awry – Three Common Vulnerabilities

## Path Traversal Attack – Backend/Middleware

The first type of attack we'll study mostly applies to the backend or middleware part of a web application – code that runs on the web server itself. This vulnerability exploits the way in which web servers often directly serve and process *files* that reside on the server. At a basic level, a website or web application often lives within a specific *directory (folder)* on the server. The main directory is the website root, from which all other file paths are derived. Say for example your website files reside in `/var/www/site/`, and your domain is `site.com`. When you navigate to the `site.com` URL, the web server will generally server a file in that folder called `index.html`, executes code from a backend application designed to handle the `/` route.

Subsequent files like an about page `about.html` are accessed using a URL like `site.com/about.html`, which tells the server to look for the `about.html` file within the root folder (`/var/www/site/about.html`). A backend application that dynamically outputs pages for a URL might execute a different Python function for a URL route, such as a query to a database (`site.com/data`). But there can be a serious flaw with simply accessing files by a given URL path.

In most filesystems, there's special *relative path* characters, that allow a user to navigate the folder structure without using full paths every time. In particular, the `./` specifier tells the computer to move *forward* into subdirectories from the current working directory. The ../ specifier tells the computer to move backward. So if you're in a folder called `/var/www/site/` that contains a subdirectory called `employees`,

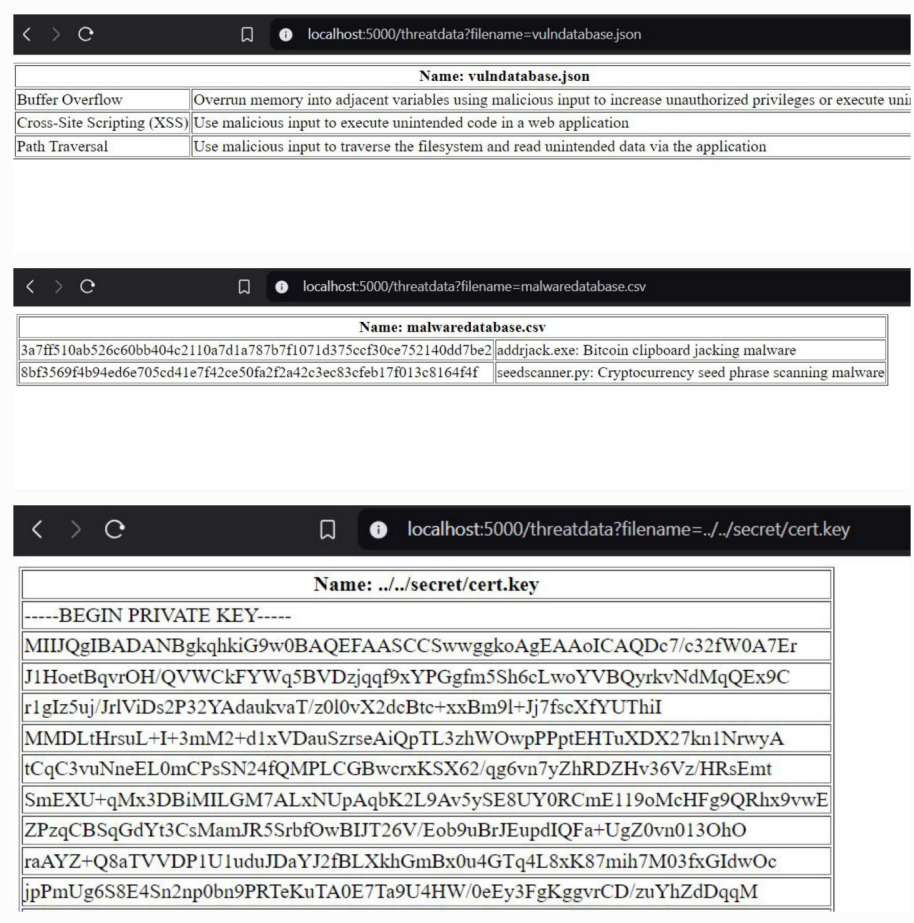
`./employees/ceo.html` can be used to operate on the ceo.html file, without having to specify the full `/var/www/site/employees/ceo.html` path. Now let's say you have a folder above the webroot called `/var/www/internal` containing some proprietary information about the company's products. You could access that using `../internal/` instead of the full path. Uh-oh...you might already be seeing how this might cause security problems.

Our first vulnerability example is a project called PathProwler. This is a Python web application demo with a path traversal vulnerability. The real function of this application is to fetch and display some simple software vulnerability and malware signature databases. Using the application legitimately, a user can navigate to a specified database file, and the Python web application will fetch and format the database in the web browser. Using the `/threatdata?filename=vulndatabase.json` URL will display some info about common software vulnerabilities from a JSON file, and `/threatdata?filename=malwaredatabase.csv` will show some malware hashes from a CSV file(from other fun chaintuts demos).

The Python code powering this demo simply takes the database filename as a GET parameter, opens the file, and processes it for display. But this is where the serious path traversal flaw comes into play.

```
path = f"{RES_DIR}{datafile}"

        if path.endswith(EXT_CSV):
            with open(path) as f:
                data = f.readlines()
        elif path.endswith(EXT_JSON):
            with open(path) as f:
                data = json.load(f)
        else:
            with open(path) as f:
                data = f.readlines()
```

Python's open function easily handles relative paths, from wherever the application is running. So what happens if we give it a backwards-moving relative path, and try to access something *outside* the web root? This vulnerable code totally allows that. For this demo, I included what should be a secret file – the web server's TLS certificate private key! If you specify the filename URL with parameter `/threatdata?filename=../../secret/cert.key`, the application will happily dump the encoded certificate private key for you on the screen. How...nice (for an attacker!)




PathProwler demonstrating a path traversal vulnerability, allowing the user to dump the server's TLS certificate private key alongside legitimate databases

This vulnerable demo is our first look at how dangerous unsanitzed input can be. In this case, the developer's failure to properly handle paths supplied by the user results in the ability to dump the server's TLS certificate private key, which is *super, duper secret*. An attacker that gains this can decrypt traffic between the server and any visitors of the site...yikes.

A simple fix for this for typical file-based web server operation is to configure your web server (such as Apache) to only allow access to files inside the web server. For applications that process URL routes directly (like PathProwler) you should write your code to sanitize the inputs – for example, only allowing a specified list of potential paths (an allowlist). For example, in PathProwler, code that only allows the known databases `vulndatabase.json` and `malwaredatabase.csv` as possible values. If your list of possible URLs is too unwieldy for an allowlist, consider code that checks that the ultimately constructed path always contains a certain base directory, like the web root. That way, any path constructed to other sensitive directories on the filesystem will be disallowed.

## Cross-Site Scripting (XSS) – Frontend

The next type of vulnerability we'll discuss occurs on the frontend of web applications – in the JavaScript code your web browser runs. This one exploits the relative flexibility of JavaScript, HTML, and how web browsers execute code within pages. Cross-site scripting is a form of code

injection attack, where some input is not properly sanitized, causing the web browser to execute the attacker's malicious code and not just the code intended by the developers.

First, it's helpful to understand how scripts execute in web application frontends. A webpage contains HTML, the markup language, which defines the page's content. CSS (style sheets) help define the look and feel of the site, but those aren't particularly relevant here. JavaScript files, or scripts written within the HTML, define how the page behaves. JavaScript is flexible and fairly powerful, and allows all sorts application features in the modern websites we use.

JavaScript is inserted into web pages via the HTML `<script>` tag. Script tags can reference a Javascript file on a web server, or can directly contain JavaScript code. Normally, users just run whatever JS code is provided by the web server. However, anything sent to your web browser can be modified, since it's on *your* computer and you have direct access to the scripts inserted in the page. Normally, this ability is harmless. It doesn't matter if you modify scripts in your web browser because it only affects code that runs on your computer. But what if we can exploit a way to get someone else's web browser to run malicious code? This is where *cross-site scripting* comes in.

Our code demo for this attack is XSSAddr. This simple web application takes a userid in the GET parameters of the URL, and fetches a user's Bitcoin deposit address to display on the page. The key function in our demo is a simple placeholder, that for our purposes, just returns a hard-coded Bitcoin address which will be inserted into the HTML of the page. Note: don't use this address for anything – it's a throwaway and the key is unavailable.

```
function generateAddress(userId)
{
     var address = "1Nzo1mojzMSKpEdWMkusZJd4arcHU6ZeH7";

     return address;
}

function fetchAndSetAddress()
{
     var userId = document.getElementById("userid").value;

     var address = generateAddress(userId);
     setAddressInDocument(address);
}
```

We're going to exploit the fact that the application gets the userid from the URL GET params (`xssaddr.html?userid=`) to execute malicious code. The code simply takes the userid, and places it in the document for later use, without performing any kind of validation or sanitization. The problem here is the fact that we can include *any* value, including HTML and JavaScript that the page will actually interpret as if it was legitimate code and not just a numerical value. JavaScript and HTML are very flexible in that way – powerful and useful for web applications, but also dangerous.

A legitimate input to the `userid` field might be `123`. But what happens if we put a script in that field? The example included in the demo looks like this, first in human readable form and then in URL-encoded form:

```
<img
src="
AfFcSJAAAAAXNSR0IArs4c6QAAAA1JREFUGFdjYGRg/
A8AAQoBAj0TXDIAAAAASUVORK5CYII=" onload="generateAddress =
function(userId){return
'1maliciousaddresskusZJd4arcHU6ZeH7';}" />

%3Cimg%20src%3D%22data%3Aimage%2Fpng
%3Bbase64%2CiVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAAAXNSR
0IArs4c6QAAAA1JREFUGFdjYGRg%2FA8AAQoBAj0TXDIAAAAASUVORK5CYII%3D
%22%20onload%3D%22generateAddress%20%3D%20function%28userId
%29%7Breturn%20%271maliciousaddresskusZJd4arcHU6ZeH7%27%3B%7D
%22%20%2F%3E
```

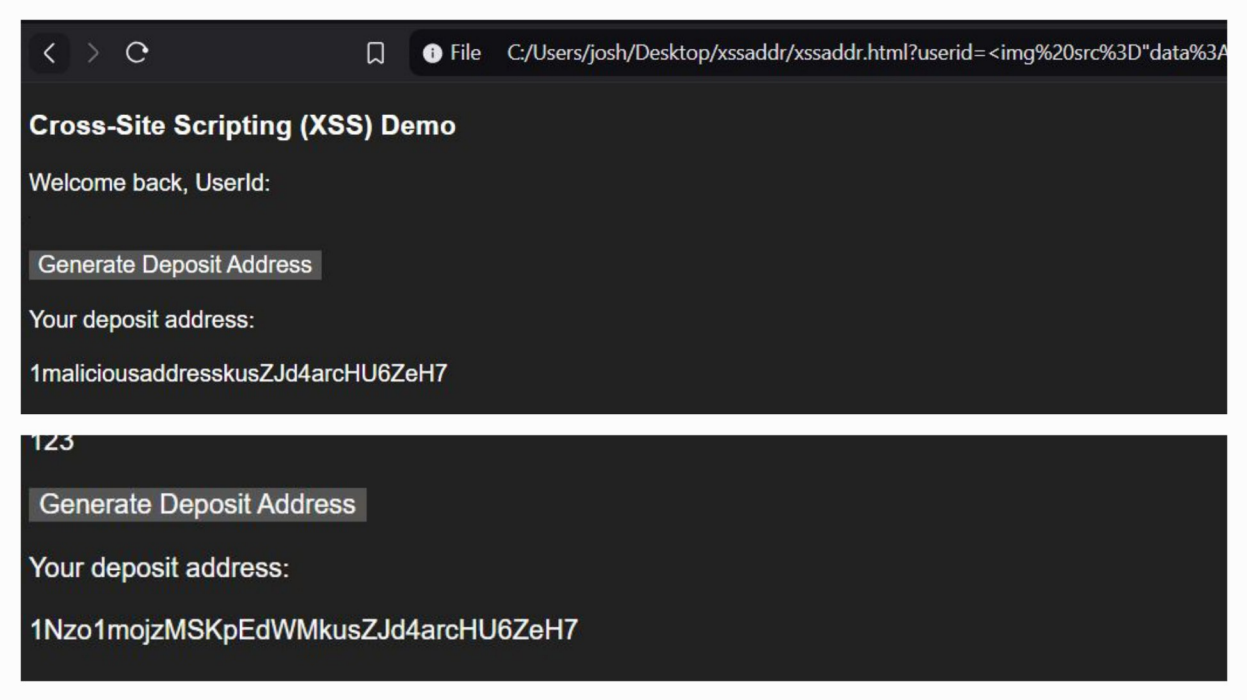
This code does a few things, that will allow it to execute upon insertion into the page. The HTML tag is an `<img>` tag, which allows the insertion of images into a page. The image data is an unobtrusive 1x1 black pixel, encoded right in the HTML rather than fetched from a remote source. The real important part is the `onload=` portion. This HTML feature specifies JavaScript code to run when the page loads the image. In this case, it sets the value of the `generateAddress` function to be a new function that returns the malicious attacker address every time, instead of the real address for the user.

Why the `<img>` tag instead of `<script>`? Well, this is a quirk of modern web application security. The possibility of XSS attacks is so prevalent and dangerous that modern JavaScript standards prohibit the execution of script tags when a value is set in the page using a DOM object's `innerHTML` attribute. So you can't simply place a script tag inside a div, for example, using innerHTML and then expecting the script to execute. But, a way around this is to exploit the legitimate uses of `onload` functions for other objects like images. We simply insert an image, and specify the JS to run when it loads. The browser is happy to comply with our request.

This is the main part of our XSS attack. When the user navigates to the URL with the userid= field containing our malicious payload, it replaces the legitimate generateAddress function with a malicious one. So when a user clicks to get their deposit address, the page will not return theirs. It will instead replace the address with the malicious attacker address every time! So an unsuspecting user might send all of their Bitcoin to the attacker wallet instead of theirs, an irreversible theft. That is a bad vulnerability for a crypto application to have!

On the surface, this vulnerability might not seem so bad. After all, who the heck is going to replace their userid with this malicious payload by accident, and how could you even convince a user to do so manually? Our security-scary friend *phishing* can be of assistance here. We get emails with links in them all the time, from seemingly legitimate-looking sources. And link *text* displayed in a document or email doesn't have to match *the actual link*. It's perfectly easy to

create a link that says https://mycryptowebsite.com/userid=123 and have the underlying link set the userid= field with our malicious payload. What's worse is that this attack exploits a flaw in the *legitimate site*. So even if a user is careful about phishing, they are less likely to be suspicious when they see the link leads to the correct site in their browser...all while getting their wallet drained.




XSSAddr showing a cross-site scripting vulnerability, displaying a malicious address instead of the user's legitimate one

Prevention of XSS attacks, again, has to do with sanitizing inputs rather than just acting on whatever the user gives. First, perform validation based on expected inputs. In our example, checking that a userid is numeric and rejecting a non-numeric value can help prevent this attack. As well, use escaping – which takes the input characters and ensures that the web browser interprets them as literal text instead of executable code. In our example this would mean the malicious input would show on the page with the literal text snippets `<img src=`, `generateAddress=`, etc. instead of the browser acting upon the HTML and JavaScript specified.

## Buffer Overflow Attacks – Low-Level/Application Software

The final type of attack we will discuss is one that's been around for nearly as long as computing itself – the buffer overflow attack. This attack occurs in low-level code that acts directly on segments of memory without safety checking. Most commonly, these vulnerabilities occur in code written in C and C++. I say low-level because C and C++ are compiled down to machine code, and allow the direct addressing of memory segments in a way that higher-level languages

do not. But the software could be anything from an operating system to a user application or command-line tool.

A buffer overflow occurs when the code places bytes in a segment of memory that literally *overflow* into the next segments of memory, intended to store another code variable. An application might allocate eight bytes worth of memory for an eight character string, for example. Next door in memory might be a one byte short integer, or a double, or another string. The buffer overflow occurs when the code mistakenly places more bytes than allocated into memory, thus overwriting some other variable's data. In this simple scenario, trying to write a nine-character string into the eight-character buffer would result in the ninth character overwriting whatever value was supposed to be stored for the one byte short integer. This can make your code do all sorts of wacky things, from simply crashing to introducing very serious security vulnerabilities.

Our code demo for this vulnerability is OminousOverflow, a command-line program written in C. This program is designed to show a user a simple secret Bitcoin private key if and only if they authenticate using a correct password. However, the structure of this code has a fatal flaw, a buffer overflow that interferes with the authentication process. This particular flavor of buffer overflow is a *privilege escalation* attack, allowing us to view a secret without the proper authentication.

The code contains a struct containing several variables for a user and their authentication data:

```
struct AuthUser
{
          int id;
          char secret[53];
          char credential[9];

          char credential_cached[8];
          short authenticated;
};
```

Of the most importance to understanding the buffer overflow is the eight-byte char array for the cached password, and right next to it in memory is a one-byte short integer storing whether or not the user is authenticated. In C, a value of 0 is generally interpreted as "false", and a non-zero value (usually 1) is considered "true".

The function that checks the password and sets the authenticated byte is as follows:

```
short authenticate(struct AuthUser* auth_user, char*
credential_check)
{
     if (strcmp(credential_check, auth_user -> credential) == 0)
     {
          auth_user -> authenticated = 1;
     }
```

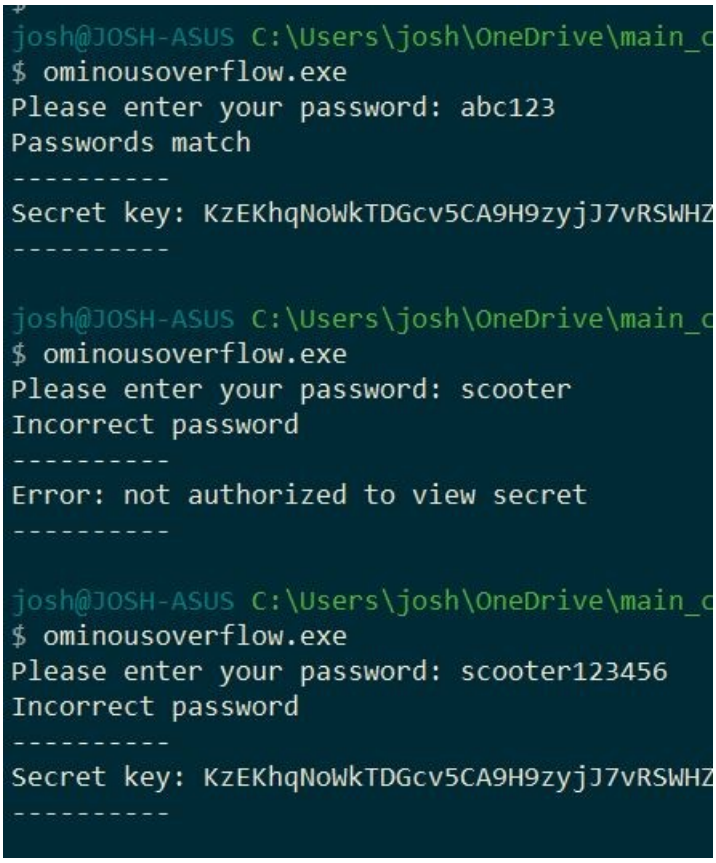
```
    else
    {
        auth_user -> authenticated = 0;
    }

    strcpy(auth_user -> credential_cached, credential_check);
}
```

This function sets the authentication byte, and also stores a copy of the password in the AuthUser struct. This is likely not a great practice to begin with, but serves as a somewhat reasonable example of a potential bad coding practice that also enables our buffer overflow attack.

Another key part of this vulnerability is the nature of the C standard library's `strcpy` function. This function relies on C-strings (character arrays) containing a null-terminator character '`\0`' to know when to stop copying bytes. The strcpy function does not do *bounds checking* on the destination array. If you give it a nine-character null-terminated string and specify an eight-character destination array, it is more than happy to overwrite whatever byte happens to be next door. Technically, the behavior of strcpy when you give in an insufficiently-sized destination string is undefined. But in many cases, it will happily overwrite additional slots of memory without necessarily crashing the program. And that's exactly what happens in our demo, with some even more disastrous consequences than simply weird behavior.

Remember our struct that contains the cached credential with the authenticated byte next to it? If we supply the correct less-than-eight character password, the program stores that password and the correct authentication byte true (1) no problem. We're then allowed to view our secret Bitcoin key. If we give an incorrect eight-or-less character password, the program correctly rejects our attempt to authenticate and stores false (0). We will not be permitted to view the secret key. But giving an incorrect, greater-than-eight character password causes something very interesting to happen. In this case, the code will write all nine or more characters to memory, filling up the eight available spaces in the `credential_cached` buffer, and then *continuing* into the space occupied by `authenticated`.

This is the crux of our buffer overflow attack. Since we've now overwritten the authenticated byte with a non-zero value, our program allows us to view the secret Bitcoin key *with an incorrect password.* So, we don't need to know the right password at all – we can trick the program into showing us the secret with *any* eight, ten, twelve, character password – a *privilege escalation* attack!



OminousOverflow demonstrating a privlege escalation buffer overflow attack, where a secret key is revealed using a longer-than-expected, incorrect password

Preventing buffer overflows can involve several coding practices, and can sometimes be tricky. It's often hard to spot these kinds of vulnerabilities, because it's not immediately obvious looking at the code that a malicious input can overwrite some other variable in memory. A good first step is to always perform some sort of *bounds checking*, and use functions that are designed to enforce bounds. For example, ensure that when you copy strings, you account for the maximum possible string size for your buffer explicitly. It also helps to use more secure functions like `strlcpy`, which requires a buffer size and always null-terminates the copied string.

Another approach, although a much bigger lift, is to avoid using languages that allow this sort of vulnerability in the first place. Rust is a compiled, lower-level language with C/++ like performance but designed with memory safety in mind. If your project doesn't have specific performance requirements that require compilation, consider using a managed or interpreted language like Python, Java, C# .NET, etc. These languages manage memory and things like string copies under the hood for you, and generally prevent these types of attacks, barring implementation bugs in the languages themselves and edge-case scenarios where unsafe code might be allowed. In general, it's far more likely you'll write a buffer-overflow vulnerability into your C/++ code than any of these alternatives.

# Insidious Inputs – Sanitize for Your Safety!

All of these fascinating vulnerabilities and examples ultimately stem from variations on a them – malicious input! As software engineers and security professionals, it's important to understand how these attacks work, and ways to mitigate them. Ultimately, it's best to always focus on validating and sanitizing the user input our code handles, to prevent malicious inputs from causing unexpected behavior. Garbage in, garbage out is certainly a truism in the world of software. Keeping up with vulnerabilities can also protect us as users of software. Although we cannot control the development process, it's interesting to know the ways in which attackers might exploit the software we use to do something malicious. Application security is fascinating, and important. So if you're just learning to code or you're a seasoned professional, sanitize those inputs and keep your software safe!