Web Application Security

Rajendra Kachhwaha
rajendra1983@gmail.com

August 5, 2015

Outline

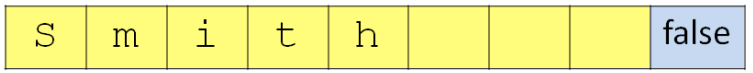
Buffer Overflow

A buffer overflow (BOF) occurs when the space allocated to a variable (typically an array or string variable) is insufficient to accommodate the variable in its entirety.
For example, a certain amount of buffer space is allocated for an array. If array bounds are not checked while populating it, the array may overflow into contiguous memory and corrupt it.
Interestingly, this could cause an attacker to subvert the normal flow of a program. Malicious code supplied by the attacker in the buffer could be executed.
To understand Buffer Overflow, we need to understand Subroutine calling conventions and Organization of the Program Stack.

Buffer Overflow

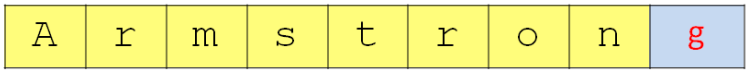
```
boolean rootPriv = false;
char name[8];
cin >> name;
```

When the program reads the name “Smith”



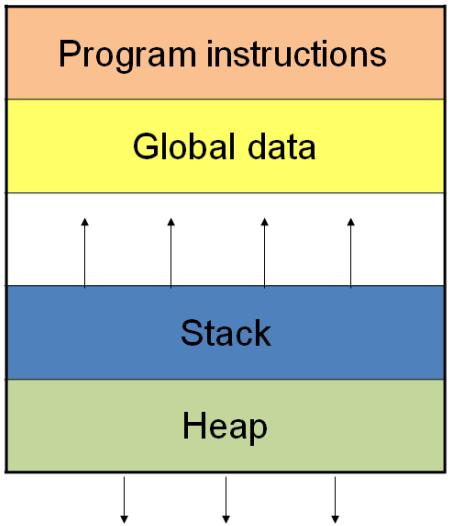
char name[8] rootPriv

When the program reads the name “Armstrong”



char name[8] rootPriv

Program Memory Organization



Buffer Overflow

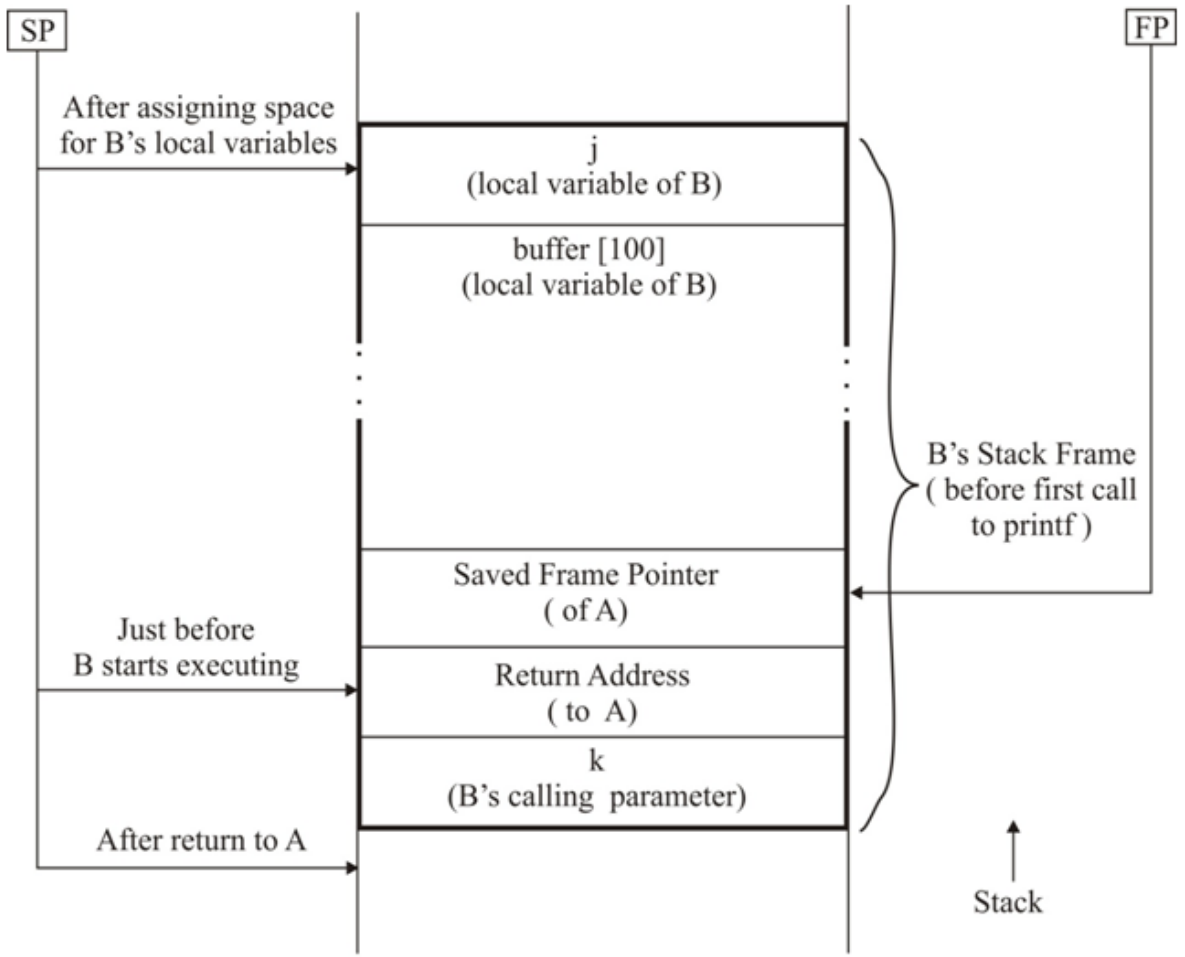
```
int A( )
{
    B(5);
    return 0;
}
void  B(int k )
{
    char  buffer[100];
    int  j = 3+k;
    printf( " Enter name: " );
    gets(buffer);
    printf(" Hello %s ", buffer);
    return;
}
```

When A calls B, the following are allocated or loaded on the stack:

- The **arguments** (parameters) used while calling B
- A's **return address**, i.e., the address at which A resumes on completion of B
- A copy of A's **Frame Pointer**
- The local or **automatic variables** of B.

(In this case, the local variable, *buffer*, is declared to be an array of 100 characters. So 100 bytes have been allocated on the stack for *buffer*.)

Buffer Overflow



Buffer Overflow

Provide input to a buffer on the stack which includes malicious code. Overflow the buffer so that the return address to the calling program is overwritten with the address of the malicious code. That way, when the called function terminates, it will not return to the calling program. Instead, the malicious code will be executed.

Overcoming BOF:

Make the stack non-executable: This prevents malicious code on the stack from being executed.

Compiler-based option: Place a "canary variable" on the stack between the local variables and the return address. If a BOF modifies the return address, the canary will be corrupted. This will be detected by the compiler and the program will be aborted.

Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys or session tokens or to exploit other implementation flaws to assume other user's identities. You may be vulnerable if:

1. User authentication credentials aren't protected when stored using hashing or encryption.
2. Credentials can be guessed or overwritten through weak account management functions (e.g., recover password, weak session IDs).
3. Session IDs are exposed in the URL (e.g., URL rewriting).
4. Session IDs dont timeout, or user sessions or authentication tokens are not properly invalidated during logout.

Broken Authentication and Session Management

Example Attack Scenarios:

Scenario 1: Airline reservations application supports URL rewriting, putting session IDs in the URL:
http://example.com/sale/saleitems?sessionid=268544541&dest=Hawaii
An authenticated user of the site wants to let his friends know about the sale. He e-mails the above link without knowing he is also giving away his session ID. When his friends use the link they will use his session and credit card.

Scenario 2: Application's timeouts are not set properly.

Scenario 3: Insider or external attacker gains access to the system's password database. User passwords are not properly hashed, exposing every user's password to the attacker.

Broken Authentication and Session Management

The primary recommendation for an organization is to make available to developers:

1. A single set of strong authentication and session management controls.
2. Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs.

Insecure Direct Object References

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data. You may be vulnerable if:

1. For direct references to restricted resources, does the application fail to verify the user is authorized to access the exact resource they have requested?
2. If the reference is an indirect reference, does the mapping to the direct reference fail to limit the values to those authorized for the current user?

Insecure Direct Object References

Example Attack Scenarios: The application uses unverified data in a SQL call that is accessing account information:

```
String query = "SELECT * FROM accts WHERE account
= ?";

PreparedStatement pstmt =
connection.prepareStatement(query , … );

pstmt.setString( 1, request.getParameter("acct"));

ResultSet results = pstmt.executeQuery( );
```

```
http://example.com/app/accountInfo?acct=notmyacct
```

The attacker simply modifies the 'acct' parameter in their browser to send whatever account number they want. If not verified, the attacker can access any user's account, instead of only the intended customer's account.

Insecure Direct Object References

How Do I Prevent 'Insecure Direct Object References'?
Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

1. Use per user or session indirect object references: This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop down list is used.
2. Check access: Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object.
3. Code review & Testing is effective for identifying direct object references and whether they are safe.

Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, servers, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date. You may be vulnerable if:

1. Is any of your software out of date? This includes the OS, Web/App Server, DBMS, applications, and all code libraries.
2. Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?
3. Are default accounts and their passwords still enabled and unchanged?
4. Does your error handling reveal stack traces or other overly informative error messages to users?

Security Misconfiguration

Example Attack Scenarios:

Scenario 1: The app server admin console is automatically installed and not removed. Default accounts are not changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario 2: Directory listing is not disabled on your server. Attacker discovers he can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which he de-compiles and reverse engineers to get all your custom code.

Scenario 3: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws.

Scenario 4: App server comes with sample applications that are not removed from your production server.

Security Misconfiguration

How Do I Prevent 'Security Misconfiguration'?
The primary recommendations are to establish all of the following:

1. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include all code libraries as well.
2. A strong application architecture that provides effective, secure separation between components.
3. Consider running scans and doing audits periodically to help detect future misconfiguration or missing patches.

Sensitive Data Exposure

Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.You may be vulnerable if:

1. Is any of this data stored in clear text long term, including backups of this data?
2. Is any of this data transmitted in clear text, internally or externally?
3. Are any old / weak cryptographic algorithms used?

Sensitive Data Exposure

Example Attack Scenarios:

Scenario 1: An application encrypts credit card numbers in a database using automatic database encryption. However, this means it also decrypts this data automatically when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text. The system should have encrypted the credit card numbers using a public key, and only allowed back-end applications to decrypt them with the private key.

Scenario 2: A site simply doesnt use SSL for all authenticated pages. Attacker simply monitors network traffic, and steals the users session cookie.

Scenario 3: The password database uses unsalted hashes to store everyones passwords. A file upload flaw allows an attacker to retrieve the password file.

Sensitive Data Exposure

How Do I Prevent 'Sensitive Data Exposure'?
The primary recommendations are to establish all of the following:

1. Dont store sensitive data unnecessarily. Discard it as soon as possible.
2. Ensure strong standard algorithms and strong keys are used, and proper key management is in place.
3. Ensure passwords are stored with an algorithm specifically designed for password protection, such as bcrypt.
4. Disable auto-complete on forms collecting sensitive data and disable caching for pages that contain sensitive data.