Prüfung 10.01.2022

Sichere Softwareentwicklung VO

Erklären Sie die Programmiersünde "Format String Problems" (Bug). Bei welcher Art von Befehlen tritt dieser auf?

Die Programmiersünde **"Format String Problems"** tritt auf, wenn unsichere oder nicht validierte Eingaben als Formatierungsstring in Funktionen wie `printf`, `scanf` oder ähnlichen in C/C++ verwendet werden. Dadurch können Angreifer spezielle Platzhalter wie `%x` oder `%n` einfügen, um Speicher auszulesen oder zu manipulieren.

Diese Bugs treten bei **Ein-/Ausgabebefehlen** auf, die Formatierungsstrings interpretieren, z. B.:

- `printf`
- `fprintf`
- `sprintf`

Sie gehören zu den Sicherheitsrisiken, die Code-Injection oder Speicherverletzungen ermöglichen.

Geben Sie ein Beispiel in Codeform für einem Format String Bug an.

Hier ist ein einfaches Beispiel für einen **Format String Bug** in C:

```
#include <stdio.h>int main() {
    char userInput[100];

    // Benutzer gibt einen Format-String ein
    printf("Geben Sie etwas ein: ");
    scanf("%99s", userInput);

    // Unsichere Verwendung des Format-Strings
    printf(userInput); // Bug: Benutzereingabe wird als For
```

```
mat-String interpretiert

    return 0;
}
```

Problem:

Wenn der Benutzer z. B. `%x %x %x` eingibt, wird der Speicherinhalt angezeigt. Noch schlimmer: Angreifer können `%n` verwenden, um Speicherwerte zu manipulieren.

Sicherer Ansatz:

Immer einen festen Format-String verwenden:

```
printf("%s", userInput);
```

Welche Konsequenzen kann ein Format String Bug haben? Nennen Sie eine mögliche Gegenmaßnahme.

Antwort:

Ein **Format String Bug** kann folgende Konsequenzen haben:

- **Informationsleakage**: Angreifer können Speicherinhalte auslesen, z. B. Passwörter oder andere sensible Daten.
- **Speichermanipulation**: Durch `%n` können Angreifer Werte an beliebigen Speicheradressen überschreiben, was zu Codeausführung (Remote Code Execution) führen kann.

Mögliche Gegenmaßnahme:

- Validierung und Einschränkung der Benutzereingaben.
- Immer einen festen Format-String verwenden, z. B. `printf("%s", userInput);`, statt Benutzereingaben direkt als Format-String zu nutzen.

Betrachten Sie den folgende Source Code:

```
#include <stdio.h>
#include <string.h>
```

```
int main(int argc, char* argv[])
{
    char input[50]; // Problem: Keine Garantie, dass 'inpu
t' am Ende einen Nullterminator hat.
    int input_lgth = strlen(argv[1]); // Problem: Kein Chec
k, ob argv[1] existiert (argc < 2 führt zu einem Absturz).
    int copy_lgth;

    if (input_lgth > 50) // Problem: Es wird nur geprüft, o
b input_lgth größer als 50 ist, aber Nullterminierung fehl
t.
    {
        copy_lgth = 50; // Gut: Begrenzung der Länge, aber
unzureichend ohne Nullterminierung.
    }
    else
    {
        copy_lgth = input_lgth; // Kein direkter Fehler, ab
er es fehlt ein Sicherungsmechanismus.
    }

    strncpy(input, argv[1], copy_lgth); // Problem: 'strncp
y' fügt keinen Nullterminator hinzu, wenn copy_lgth == 50.

    return 0; // Rückgabewert 0 ist korrekt, aber das Progr
amm meldet bei Fehlern nichts.
}
```

3. (a) Erklären Sie im Kontext von Git die Begriffe Branches und Merge.

Korrektur der Frage:

Erklären Sie im Kontext von Git die Begriffe **Branches** und **Merging**.

Antwort:

- **Branches:**

Branches sind separate Entwicklungszweige im Git, die es ermöglichen, Änderungen unabhängig vom Hauptzweig (z. B. `main` oder `develop`) vorzunehmen. Jeder Branch repräsentiert einen isolierten Arbeitsbereich, z. B. für neue Features, Bug-Fixes oder Release-Vorbereitungen.

- **Merging:**

  Beim Merging werden die Änderungen eines Branches in einen anderen integriert. Dabei können Konflikte entstehen, die manuell gelöst werden müssen. Typische Merge-Arten sind:

  - **Fast-Forward Merge:** Der Ziel-Branch wird einfach auf den Quell-Branch vorwärts bewegt.
  - **Three-Way Merge:** Es werden gemeinsame Vorfahren der Branches berücksichtigt, um die Änderungen zusammenzuführen.

---

3. (b) Ordnen Sie den Farben den zugehörigen Branch-Typ zu.

Korrektur der Frage:

Ordnen Sie die Farben in der Git-History den entsprechenden Branch-Typen nach Git-Flow zu.

Antwort:

- **Hellblau:** Release Branch (z. B. zur Vorbereitung von Versionen wie `v1.0`)
- **Violett:** Develop Branch (zentraler Entwicklungszweig, in dem neue Features zusammengeführt werden)
- **Dunkelgrün:** Feature oder Bug-Fix Branch (zur Entwicklung neuer Funktionen oder Behebung von Fehlern)

---

3. (c) Wie wird die Integrität in einem Git-Repository gewährleistet?

Korrektur der Frage:

Wie wird die Integrität eines Git-Repositorys sichergestellt?

Antwort:

Die Integrität eines Git-Repositorys wird durch folgende Mechanismen gewährleistet:

1. **SHA-1-Hashing:**

Jede Änderung (Commit) wird mit einem eindeutigen Hash (SHA-1) versehen. Dieser stellt sicher, dass Änderungen nachvollziehbar und manipulationssicher sind.

2. **Unveränderliche Commits:**

   Einmal erstellte Commits können nicht verändert werden, ohne den Hash zu ändern, wodurch Manipulationen erkannt werden.

3. **Historie:**

   Die Commit-Historie dokumentiert alle Änderungen, sodass man den Zustand des Repositorys zu jedem Zeitpunkt wiederherstellen kann.

4. **Verteilte Kopien:**

   Jeder Entwickler hat eine vollständige Kopie des Repositorys, was die Datenintegrität selbst bei Serverausfällen sicherstellt.

4. Die folgenden Fragen zielen auf das Security-Modell in Webbrowsern ab.

(a) Nehmen wir an, in einem Webbrowser sind zwei Tabs offen. In einem steht in der Adressleiste

`https://www.example.com:443/dir1/index.html`, und in der Adressleiste des anderen Tabs steht

`https://example.com/dir2/other.html`. Werden die beiden Seiten als derselbe Origin betrachtet?

Antwort: Ja, die beiden Seiten gehören zum gleichen Origin, da das Protokoll (`https`), die Domain (`example.com`), und der Port (`443`) identisch sind.

(b) Beschreiben Sie den Unterschied zwischen impliziter und expliziter Authentifizierung bei Webapplikationen.

Antwort:

- **Implizite Authentifizierung:**

  Hier erfolgt die Authentifizierung durch automatisch gesendete Informationen, wie Cookies oder HTTP-Header (z. B. `Authorization`-Header). Der Benutzer bemerkt dies nicht direkt.

- **Explizite Authentifizierung:**

  Hier erfolgt die Authentifizierung durch aktive Eingabe von Anmeldeinformationen durch den Benutzer, z. B. Benutzername und

Passwort über ein Login-Formular.

(c) Was bewirkt die Same-Origin Policy in Webbrowsern?

- **(a)** Dass beim Aufruf von Ressourcen fremder Seiten die HTTP-Antwort nicht z. B. per JavaScript wörtlich gelesen werden darf.

  Antwort: Wahr

- **(b)** Dass innerhalb einer Seite (Origin) clientseitig keinerlei HTTP-Requests zu fremden Seiten gesendet werden können.

  Antwort: Falsch

- **(c)** Dass Webseiten keinerlei Ressourcen (JavaScript, CSS, etc.) von fremden Domänen einbinden dürfen.

  Antwort: Falsch

- **(d)** Dass Cross-site Request Forgery-Angriffe vollständig verhindert werden.

  Antwort: Falsch

5. (a) Was ist der Unterschied zwischen Authentifizierung und Autorisierung?

Antwort:

- **Authentifizierung**: Bestätigung der Identität eines Benutzers, z. B. durch Benutzername und Passwort.
- **Autorisierung**: Überprüfung, ob ein authentifizierter Benutzer berechtigt ist, eine bestimmte Aktion durchzuführen oder auf bestimmte Ressourcen zuzugreifen.

(b) Was können Sie bei der Entwicklung einer Webapplikation gegen das Problem „unsichere direkte Objektreferenzen“ tun? Es gibt genau eine richtige Antwort.

- **(a)** Die Zugehörigkeit des aufgerufenen Objektes zum/zur aktuell angemeldeten Benutzer*in überprüfen.

  Antwort: Wahr

- **(b)** Bei jeder Anfrage überprüfen, ob das aktuelle Konto die entsprechende Rolle hat, um die Funktion aufzurufen.

Antwort: Falsch

- **(c)** TLS für die Transportverschlüsselung einsetzen.

  Antwort: Falsch

- **(d)** Aufsteigende Objekt-IDs verwenden.

  Antwort: Falsch

6. (a) Welcher ist der Unterschied zwischen den Maßnahmen gegen Reflected Cross-Site Scripting und Stored Cross-Site Scripting?

- **(a)** Bei Stored XSS passiert die Ausgabekodierung in der Datenbank, bei Reflected XSS bei der Ausgabe.

  Antwort: Falsch

- **(b)** Bei Stored XSS ist die Ausgabekodierung unabhängig vom jeweiligen Ausgabe-Kontext.

  Antwort: Falsch

- **(c)** Es gibt keinen grundsätzlichen Unterschied.

  Antwort: Wahr

- **(d)** Whitelisting von Eingabeparametern ist bei Stored XSS wirkungslos.

  Antwort: Falsch

7. (a) Wie funktionieren, generisch gesprochen, Injection-Angriffe, und zwar unabhängig von der Technologie (SQL, SMTP, LDAP, etc.)?

Antwort:

Injection-Angriffe funktionieren, indem ein Angreifer speziell präparierte Eingaben einschleust, die von der Anwendung fälschlicherweise als legitime Befehle oder Daten interpretiert werden. Dadurch können unerwünschte Aktionen ausgeführt werden, wie das Auslesen oder Manipulieren von Daten oder das Ausführen von Befehlen auf dem Server.

---

7. (b) Nehmen wir an, es gebe eine OS-Command-Injection-Lücke in einer Webapplikation, die erfolgreich ausgenützt wird. Im Kontext welches Betriebssystembenutzers werden die injizierten Kommandos allgemein gesprochen ausgeführt?

- **(a)** Mit dem root-Benutzer.

  Antwort: Falsch

- **(b)** Als privilegierter Betriebssystembenutzer.

  Antwort: Falsch

- **(c)** Als jener Benutzer, unter dem der Webserver bzw. Applikationsserver läuft.

  Antwort: Wahr

- **(d)** Als nichtprivilegierter Benutzer.

  Antwort: Falsch