Softwaresicherheit

Jens Willmer[1] und Tobias Maier[1]

DHBW Stuttgart Campus Horb, Florianstraße 15, 72160 Horb am Neckar

Diese Arbeit befasst sich mit Methoden der sicheren Software-Entwicklung (am Beispiel des Microsoft Security Development Lifecycles) sowie bekannten Schwachstellen in Software wie Buffer Overflows oder String Format Attacken.

I. Security Development Lifecycle

Der Security Development Lifecycle (SDL, deutsch: „Entwicklungszyklus für vertrauenswürdigen Computereinsatz“) ist ein Konzept von Microsoft, das die Entwicklung von sicherer Software konsequent vorantreiben soll. Der SDL umfasst eine Menge nützlicher Tools und Best Practices, die Entwickler bei ihrer Arbeit anwenden können. Er ist in sieben Phasen eingeteilt, die nachfolgend kurz vorgestellt werden.

A. Training

Die Voraussetzung für eine effiziente Implementierung des SDL ist ein umfassendes Training der Entwickler. Es sollte die wesentlichen Inhalte aller weiteren Phasen enthalten.

B. Requirements

Im Bereich der Requirements (Anforderungsanalyse) sollen zunächst die minimalen Anforderungen an Sicherheit und Datenschutz aufgestellt werden. Dabei gilt es zu definieren, welche Daten besonders schutzbedürftig sind. Abhängig von diesen Anforderungen können spätere Sicherheitslücken in der Software leichter kategorisiert und priorisiert werden. Anschließend kann eine sogenannte Bug Bar festgelegt werden. Das bedeutet, es wird genau definiert, zu welchem Zeitpunkt maximal welche Anzahl von Sicherheitslücken vorhanden sein darf. Beispielsweise ist es sinnvoll, dass beim Release der Software keine als „kritisch“ eingestuften Lücken mehr existieren.

Für einzelne Funktionen einer Software kann es außerdem nötig sein, detailliertere Risikobewertungen durchzuführen, um als Entwickler zu verstehen, wie wichtig die Sicherheit einer entsprechenden Stelle ist.

C. Design

In der Modellierungsphase gilt es unbedingt den Grundsatz „Security by Design“ zu beachten. Die sicherheitsrelevanten Erkenntnisse aus der Anforderungsanalyse sind hier auch im Design der Software umzusetzen. Es ist deutlich günstiger und einfacher, bereits dem Entwurf eines Programms entsprechende Sicherheitsmechanismen vorzusehen, als diese später nachträglich zu implementieren.

Dabei müssen potentielle Angriffsflächen so weit wie möglich reduziert werden, möglichst ohne dabei den Leistungsumfang der Software einzuschränken. Das Prinzip der geringsten Rechte – dabei bekommt jeder Benutzer oder jede Funktion nur genau die Rechte, die sie unbedingt braucht – sollte bereits beim Entwurf konsequent angewandt werden. Das sogenannte „Threat modelling“ (Bedrohungsanalyse) beschreibt einen Prozess, bei dem während der Entwurfsphase bereits getestet wird, wo die Schwachstellen des Systems liegen, um diese dann anschließend zu beseitigen.

D. Implementation

In der Implementierungsphase geht es darum, nur sichere Funktionen einer Programmiersprache zu verwenden. Darauf müssen die Entwickler einerseits selbst achten, andererseits können sie durch Tools dabei unterstützt werden. Wichtig ist, den entstandenen Quellcode während der Implementierung regelmäßig durch Reviews oder mit Hilfe von Software zu überprüfen und hinsichtlich Sicherheitslücken zu analysieren.

E. Verification

Die Testphase dient dem Testen des laufenden Programms, also nicht mehr auf Codeebene. Dabei müssen Themen wie Speicherplatzbehandlung, Benutzerrechte, usw. durch geeignete Testverfahren abgedeckt werden. Das Testen orientiert sich dabei an den Anforderungen an die Software. Eine besondere Form des Testens ist das sogenannte „Fuzz Testing“. Dabei wird versucht, die Software durch zufällig erzeugte oder gezielt bösartige Eingaben anzugreifen. Darüber hinaus ist es wichtig, in dieser Phase auf die Schwachstellen aus der Design- und der Implementierungsphase zurückzugreifen. Es muss getestet werden, ob die entsprechenden Gegenmaßnahmen tatsächlich greifen, und ob dadurch eventuell neue Sicherheitslücken entstanden sind.

F. Release

Im Zuge des Releases einer Software sollte ein Plan aufgestellt werden, der die entsprechenden Ansprechpartner bei Sicherheitsproblemen festlegt. Es sollte ein finales Sicherheitsreview durchgeführt werden, das von der Anforderungsphase bis zur Testphase alle erkannten Probleme enthält und erneut testet. Anschließend ist es wichtig, sämtliche Erkenntnisse, Dokumente, Quellcodes usw. umfassend zu archivieren.

G. *Response*

Diese Phase beginnt im Anschluss an das Release einer Software und umfasst alle sicherheitsrelevanten Themen, die während des Betriebs auftreten. Tauchen dabei neue Sicherheitslücken auf, müssen diese durch Updates behoben werden.

II. BUFFER OVERFLOW

Buffer-Overflow oder auch Speicherüberlauf ist eines der am meisten auftretenden Sicherheitsprobleme in der Softwareentwicklung. Es wird von Angreifern dazu genutzt, in das betroffene System zu gelangen.

Verantwortliche für diese Schwachstelle ist in erster Linie der Entwickler, welcher Daten aus externen Quellen vor der Verarbeitung im System nicht auf ihre Länge überprüft. Vernachlässigt der Entwickler die Überprüfung der Daten und schreibt diese in einen Speicherbereich, welcher zu klein für die Datenmenge ist, läuft dieser über und es entsteht der so genannte Buffer-Overflow.

Durch den Speicherüberlauf überschreibt die eingefügte Datenmenge dabei den Inhalt anderer Felder mit ihrem eigenen. Dadurch ist es einem Angreifer möglich, an bestimmte Stellen im Speicher eigene Daten zu hinterlegen, womit er eigenen Programmcode einschleusen kann, um die Funktionsweise des Programms zu verändern.

Sehr anfällig für einen Buffer-Overflow sind Sprachen wie C und C++, da hier C-Standartfunktionen verwendet werden, welche keine eingebaute Überprüfung beinhalten. Doch selbst interpretierte Sprachen sind von Buffer-Overflows nicht gefeit, es genügt eine entsprechende Sicherheitslücke im Interpreter.

Um besser verstehen zu können, wie diese Schwachstelle ausgenutzt werden kann, wird nachfolgend die Speicherverwaltung eines modernen Betriebssystems betrachtet. Anschließend werden zwei Anwendungsbeispiele präsentiert, die einen Buffer-Overflow ausnutzen.

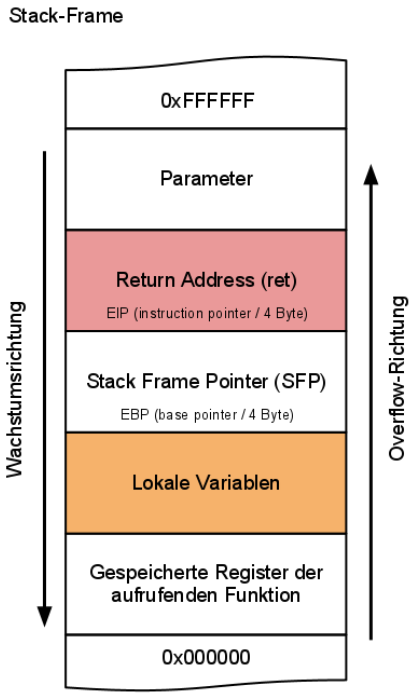
A. *Speicherverwaltung*

1) *Memory Management Unit*

Um zu verhindern, dass laufende Programme sich gegenseitig in den Speicher schreiben, wird jedem Programm ein virtueller Adressraum zugewiesen. Die Memory Management Unit (MMU, Speicherverwaltungseinheit) ist dafür zuständig, die virtuellen Adressen in physikalische Adressen zu übersetzen. Der virtuelle Adressraum ist hierbei in Stack, Heap und einen Codebereich aufgeteilt.

2) *Stack*

Der Stack fängt am oberen Ende des Speichers an und wächst nach unten. Der Stack funktioniert nach dem Last-In-First-Out-Prinzip (LIFO). Hierbei können neue Elemente nur oben auf den Stapel gelegt werden und können auch nur von dort wieder heruntergenommen werden. Im Stack werden dynamische Variablen und Rücksprungadressen von Funktionen gespeichert. Desweiteren dient er als schneller Zwischenspeicher für Parameterübergaben und Daten. In Abbildung “Stack-Frame” ist der typische Aufbaue eines Stack-Frames (Stack-Rahmen) skizziert.

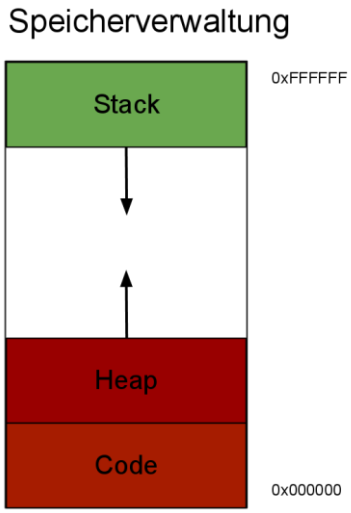


3) *Heap*

Der Heap ist eine abstrakte Datenstruktur, in der Elemente oder Objekte gespeichert werden und daraus auch wieder entnommen werden können. Der hierfür benötigte Speicher kann dazu dynamisch angefordert werden. Der Heap dient somit vorrangig der Speicherung von Mengen. In ihm lagern Felder (Arrays), Tabellen sowie extern und static Variablen. Der Heap wächst hierbei nach oben und besitzt den Vorteil, dass er anders als der Stack nicht nach dem LIFO-Prinzip funktioniert.

4) *Codebereich*

Im Codebereich wird der Quellcode des Programms in Form von Maschinenbefehlen gelagert, welche vom Prozessor zyklisch eingelesen und verarbeitet werden. Dieser Bereich lässt sich nicht manipulieren, besitzt eine feste Größe und ist vor dem überschreiben geschützt.



B. Anwendungsbeispiele

Die hier aufgeführten Anwendungsbeispiele nutzen jeweils einen Stack-Overflow aus, um entweder Variablen oder Rücksprungadressen zu manipulieren. Prinzipiell sind diese Angriffe auch auf den Heap anwendbar. Der Heap speichert allerdings andere Daten als der Stack, es kommt also darauf an, welche Daten der Angreifer manipulieren möchte.

1) Überschreiben einer Variablen

In dem unten aufgeführten Beispielprogramm wird eine Benutzeranmeldung auf ein System simuliert. Bei Eingabe des richtigem Anwendernamens und des dazugehörigen Passwortes wird die Variable *authentification* auf eins gesetzt. Im weiteren Programmablauf wird überprüft, ob die Variable gesetzt wurde, und wenn ja, wird dem Benutzer der Zugriff auf das System gewährt.

Wie dem Quellcode zu entnehmen ist, werden als Erstes die Variablen *authentification*, *cUsername* sowie *cPassword* initialisiert. Dabei wird für die Variablen *cUsername* und *cPasswort* jeweils ein 12 Byte großer Speicherbereich reserviert, da das Dateisystem Datenmengen in 4 Byte großen Blöcken abspeichert. Für die Variable *authentification* werden 4 Byte reserviert, da ein Integer standardmäßig 4 Byte im Speicher belegt.

Schreibt nun ein Angreifer einen Anwendernamen, der größer als 12 Zeichen ist, in das Feld für den Benutzernamen, löst er einen Pufferüberlauf aus, welcher die darauffolgenden Zeichen in den Speicherbereich über sich schreibt, womit er den Wert der Variable *authentification* verändert.

Da das Programm lediglich abfragt, ob *authentification* gesetzt wurde, reicht dies aus, um Zugriff auf das System zu erlangen.

```c
int main()
{
    // integer allocates 4 Bytes in memory
    int authentication = 0;

    // 12 Byte in memory because we have
    // 4 Byte adressblocks!
    char cUsername[10];
    char cPassword[10];

    printf(" Enter Username: ");
    gets(cUsername);

    printf(" Enter Password: ");
    gets(cPassword);

    if(strcmp(cUsername, "admin") == 0
       && strcmp(cPassword, "adminpass") == 0)
    {
      authentication = 1;
    }

    // checks if variable have value!
    if(authentication)
    {
        printf(" Access granted\n");
    }
    else
    {
        printf(" Wrong username and password\n");
    }

    return 0;
}
```

2) Verändern einer Rücksprungadresse

In diesem Beispiel wird der Benutzer um eine Eingabe gebeten, welche ihm daraufhin vom Programm wieder zurückgegeben wird. Das Programm ruft dazu eine Unterfunktion auf, die als erstes einen 12 Byte großen Bereich für die Benutzereingabe reserviert, daraufhin die Benutzereingabe in diesen Speicherbereich schreibt, um schließlich den Inhalt der aufrufenden Funktion zurückzugeben. Diese wird daraufhin dem Benutzer auf dem Bildschirm ausgegeben.

Zusätzlich beinhaltet dieses Programm noch eine weitere Funktion (*PrivateFunction*), die aber nicht ausgeführt werden soll und auch keine Verbindung zum Hauptprogramm hat. Das Ziel des Angreifers ist es diesmal, genau diese Funktion auszuführen. Dafür benutzt er ein speziell für diesen Zweck geschriebenes Programm, einen so genannten Exploit.

Der Exploit füllt als erstes die Variable der Unterfunktion, um daraufhin an die Position der Rücksprungadresse zu gelangen. Diese überschreibt er mit der Startadresse der privaten Funktion, welche er im Voraus über einen Debugger in Erfahrung gebracht hat.

Durch das Verändern der Rücksprungadresse wird bei Beendigung des Unterprogramms an die vom Angreifer angegebene Adresse gesprungen, womit die private Funktion ausgeführt wird und der Angreifer sein Ziel erreicht hat.

```c
void PrivateFunction(void);
const char * const RepeatFunction(void);

int main()
{
    printf("You wrote: %s", RepeatFunction());
    return 0;
}

const char * const RepeatFunction(void)
{
    char buf[12];

    printf("Write something: ");
    gets(buf);

    return strdup(buf);
}

void PrivateFunction(void)
{
    printf("--> Output of the private function!");
    exit(0);
}
```

Figure 1. 1 Opfer

```c
void main()
{
    // fill buffer and base pointer
    // 12 Byte (buffer) + 4 Byte (EBP: base pointer)
    printf("010101010101AAAA");

    // new return address (EIP: instrction pointer)
    unsigned EIP = 0x01021400;

    // overwrite EIP
    fwrite(&EIP, 1, 4, stdout);

    return 0;
}
```

Figure 2. 2 Exploit

Unter Windows (ab XP) funktioniert dieses Beispiel allerdings nicht mehr uneingeschränkt, da die MMU dem auszuführenden Programm bei jedem Aufruf einen anderen Speicherbereich zuweist. Hierdurch wird es für Angreifer erheblich aufwendiger, die richtige Sprungadresse im Exploit anzugeben.

3) Gegenmaßnahmen

Um zu verhindern, dass Systeme durch Pufferüberläufe kompromittiert werden, müssen jegliche Eingaben aus externen Quellen validiert werden. Desweiteren sollten Funktionen, die vor dem Schreiben in den Speicher nicht überprüfen, ob der Speicher überhaupt die benötigte Datenmenge reserviert hat, gemieden werden.

Weitere Möglichkeiten, die Anfälligkeiten des Systems zu minimieren, bestehen darin, den Quellcode vor der Veröffentlichung von einem anderen Programm auf die Verwendung von problematischen Funktionen untersuchen zu lassen, sowie nach der Kompilierung des Programms, mit sogenannten Fuzzern, dem Programm zufällige Datenmengen zu übergeben, wodurch Abstürze durch Pufferüberläufe provoziert werden sollen. Stürzt das Programm aufgrund einer vom Fuzzer erzeugten Datenmenge ab, sollte diese im Debug-Modus noch einmal eingegeben werden, um den Grund des Absturzes festzustellen und ihn durch Änderungen am Quellcode zukünftig auszuschließen.

III. Format String Attack

Unter einem String-Formatierungsangriff versteht man das Ausnutzen einer Sicherheitslücke in C, die das nachfolgende Beispiel veranschaulichen soll.

```
printf(„%s“,input)
```

Die Funktion wird den Inhalt der Variablen *input* als Text ausgeben. Das sogenannte Token „%s“ gibt die Formatierung der Ausgabe an. Die Funktion *printf* lässt sich jedoch auch mit nur einem Parameter aufrufen. Handelt es sich dabei um eine ungefilterte Benutzereingabe, kann ein Angreifer dies ausnutzen, um selbst Tokens zu benutzen.

```
printf(input)
```

Würde ein Angreifer nur die Eingabe „%s“ machen, wird die printf(„%s“) versuchen, den kompletten Stack auszugeben, was dem Angreifer ungewollte Einblicke in den Speicher verschaffen könnte.

Eine einfache Abhilfe gegen diese Sicherheitslücke ist es, als Entwickler diese (oder vergleichbare) Funktion immer in der sicheren Version mit Tokens zu benutzen sowie nie eine ungefilterte Benutzereingabe weiter zu verwenden.

Quellen

Microsoft Security Development Lifecycle
http://www.microsoft.com/security/sdl/default.aspx

Jürgen Wolf - C von A bis Z
http://openbook.galileocomputing.de/c_von_a_bis_z/027_c_sicheres_programmieren_001.htm

Baran Ornarli - C++ Buffer Overflow Exploit
http://www.infernodevelopment.com/c-buffer-overflow-exploit

Mark E. Donaldso - SANS Institute InfoSec Reading Roo
http://www.sans.org/reading_room/whitepapers/securecode/buffer-overflow-attack-mechanism-method-prevention_386

Simon Hecht - Buffer Overflow Tutorial
http://www.online-tutorials.net/security/buffer-overflow-tutorial-teil-1-grundlagen/tutorials-t-27-282.html

JohannesHoh - Stack- und Heap-Overflow-Schutz bei Windows XP und Vista
http://www.slideshare.net/johanneshoh/stack-und-heapoverflowschutz-bei-windows-xp-und-windows-vista

Felix Lindner - A heap of risk
http://www.h-online.com/security/features/A-Heap-of-Risk-747161.htm