

Sicherheitsaspekte bei der C-Programmierung

EUROSEC GmbH Chiffriertechnik & Sicherheit

Tel: 06173 / 60850, www.eurosec.com

© EUROSEC GmbH Chiffriertechnik & Sicherheit, 2005

Überblick

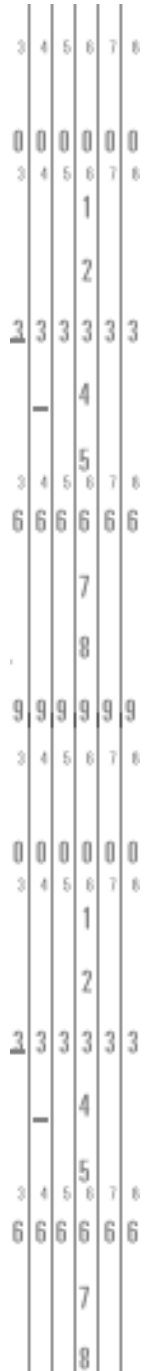
- Grundlegendes zur sicheren C-Programmierung
- Problembereiche
 - Buffer-Overflows
 - Heap-Overflows
 - Integer-Overflows
 - Aufrufe externer Programme/Anwendungen
 - Formatstring-Probleme
- Tools
 - Beispiele
- Best Practices

C und Web-Applikationen?

- "Alte" Schnittstelle: CGI
 - Heutzutage immer weniger bei Neuentwicklungen genutzt, trotzdem noch vorhanden
 - Insbesondere bei der Pflege existierender Applikationen wichtig
 - Für Web-Applikationen eine etwas andere Risikoverteilung:
 - Hohes Risiko und große Auswirkungen bei Stack-Buffer-Overflows
 - Heap-Overflows (im allgemeinen schwerer auszunutzen)
 - Formatstring-Probleme (im allgemeinen schwerer auszunutzen)
 - External Program Calls eher untergeordnet (Zugriff auf die Umgebung notwendig)
- Sicherheit von C für Web-Applikationen relevant!

C und Sicherheit

- C gilt als unsichere Programmiersprache:
 - Direkte Speicherzugriffe möglich
 - Typsicherheit, Datenkapselung nicht gewährleistet
 - Kein Sandboxing mit Policies möglich
- Größte Problem-Quelle sind Speicheroperationen:
 - Buffer-Overflows
 - Heap-Overflows
 - Fehler in der Zeigerarithmetik
- Probleme bei Speicherverwaltung lassen sich im Normalfall schwer entdecken
 - Die Verwendung von Analyse-Tools ist auch kein Allzweckheilmittel



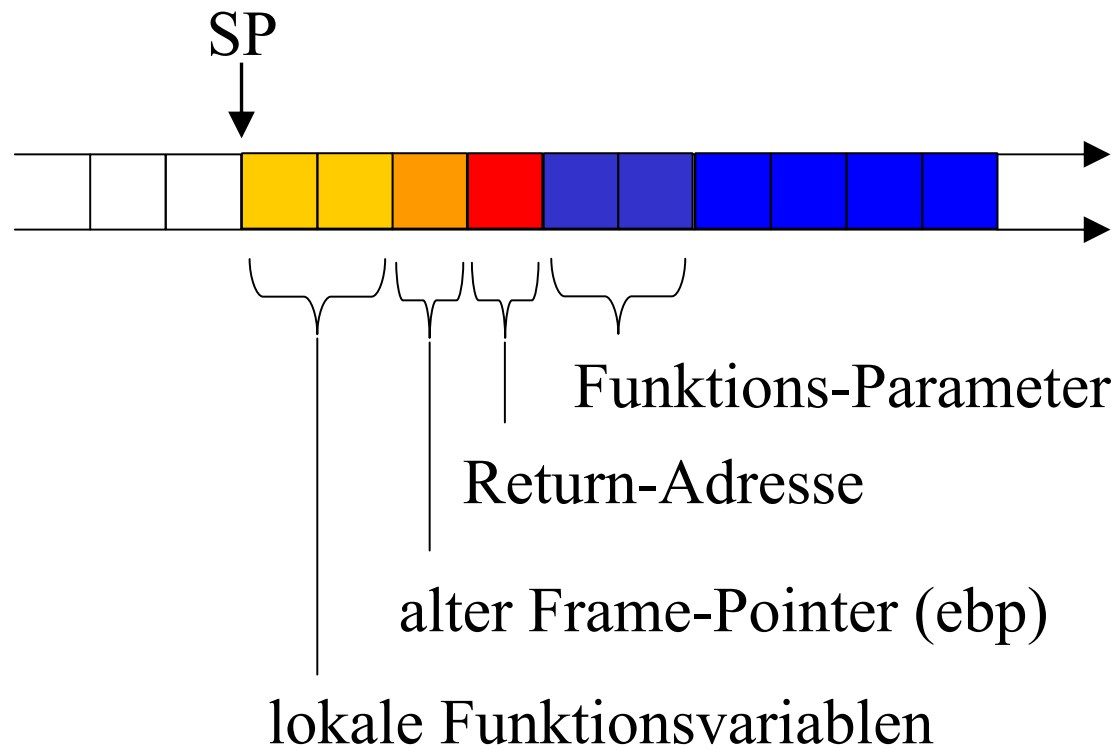
Buffer-Overflow-Probleme

Buffer-Overflows

- Ca. 50% der CERT/CC-Meldungen beziehen sich auf Buffer-Overflow-Attacken.
- Buffer-Overflow entsteht beim Missachten der Buffer-Grenzen
 - Zu wenig Platz für zu viele Daten
 - C bietet keine integrierten Mechanismen zum Überprüfen/Einhalten der Grenzen
 - Prüfung ist Aufgabe des Programmierers
- Bei automatisch oder mit `alloc()` angelegten Daten gilt:
 - Die hinterlegte Rücksprungadresse kann überschrieben werden
 - Somit kann Programmablauf beeinflusst werden
- Buffer-Overflows resultieren in:
 - Fehlerhaftem Anwendungsverhalten (DoS) bis hin zur
 - Ausführung vom eingeschleusten Angreifer-Code

BO - Grundlagen

- Stack-Aufbau beim Aufruf einer Funktion

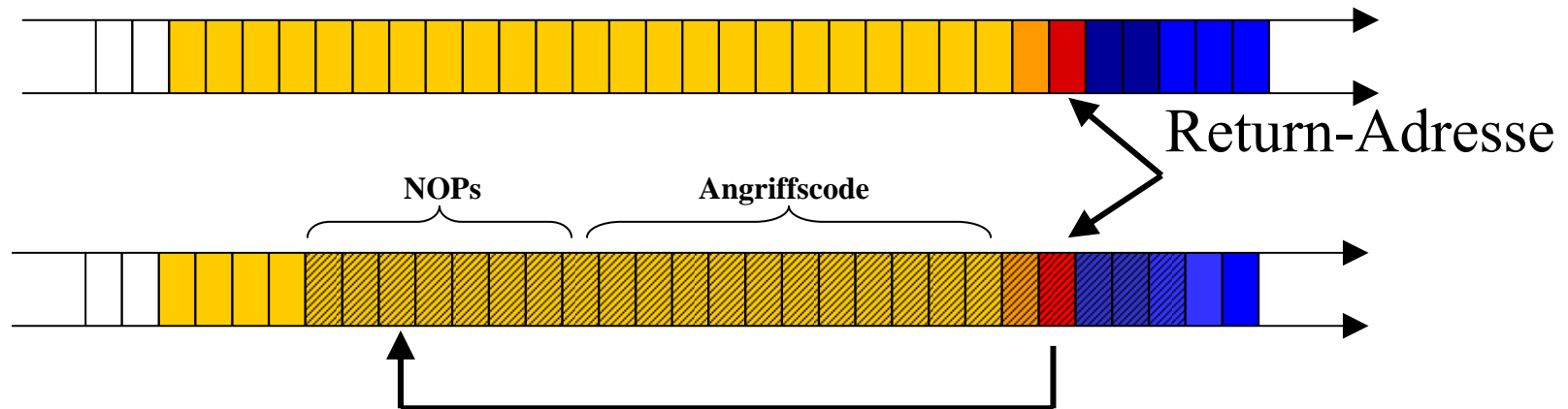


BO - Attacke

- Fehlerhafter Code:

```
int main(int argc, char *argv[]) {
    char buffer[256];
    strcpy(buffer, argv[1]);
    return 0;
}
```

- Auswirkung bei einem Exploit:



BO - kritische Funktionen

- Unsichere Funktionen nach Möglichkeit vermeiden
 - strcpy, strcat, sprintf, gets, scanf, realpath, getpass, streadd, strcpy, getopt, strtrns, getwd, select, ...

- Oder zumindest einige Vorkehrungen treffen:

Problemfall

gets

strcpy/strcat

sprintf/scanf

streadd/strecpy

realpath/getpass

Vorkehrung

Durch fgets ersetzen

Durch strncpy/strncat ersetzen (mit Längenangabe)

Precision Specifiers verwenden (z.B. "%.100s")

Vierfache Buffer-Allokierung sicherstellen

Buffer ausreichend dimensionieren

BO - Schutzmaßnahmen

- Sorgfältig programmieren
 - Benutzereingaben prüfen
 - Gefährliche Funktionen vermeiden
- Review von potentiell gefährlichen Operationen
 - Source-Code Analyse mit Tools
- Gründliches Testen unter Verwendung von Bibliotheken, die alle Speicherbereichsverletzungen aufzeigen

3	4	5	6	7	8
0	0	0	0	0	0
3	4	5	6	7	8
		1			
		2			
2	3	3	3	3	3
		4			
		5			
3	4	5	6	7	8
6	6	6	6	6	6
		7			
		8			
9	9	9	9	9	9
3	4	5	6	7	8
0	0	0	0	0	0
3	4	5	6	7	8
		1			
		2			
2	3	3	3	3	3
		4			
		5			
3	4	5	6	7	8
6	6	6	6	6	6
		7			
		8			

Heap-Overflows

Heap-Overflows

- Ursache für Heap-Overflows sind wie auch bei Stack-Overflows fehlerhafte Operationen
 - Zu wenig Platz für zu viele Daten
- Heap-Overflow erfolgt bei mit malloc() dynamisch angelegten Daten
 - Kontrollstrukturen für free() können damit überschrieben werden
 - Durch die Manipulation der Heap-Block-Strukturen kann beim Freigeben beliebige Speicheradresse mit beliebigem Wert beschrieben werden

→ Gegenmaßnahmen:

- Benutzereingaben prüfen (Längenprüfung)
- Gewährleisten, dass free() und delete() nur einmalig ausgeführt werden
 - Freigeben eines bereits freigegebenen Speichers ruft Probleme hervor



Integer-Overflows

Integer-Overflows

- Integer-Overflows entstehen durch Über- bzw. Unterschreitung eines gültigen Wertebereichs
 - Resultiert sich u.U. im Fehlverhalten der Anwendung
- Beispiel:

```
size = strtoul(argv[1], NULL, 10);
buf = alloca(size + 1);
...
ret = read(0, buf, size);
```

- Bei Verwendung von $2^{32} - 1$ werden 4GB gelesen

→ Gegenmaßnahmen

- Benutzereingaben prüfen
- Wertebereiche festlegen
- Konvertierung prüfen (insbesondere signed ↔ unsigned)



Aufrufe externer Programme

Aufrufen externer Programme

- Verwendung von Umgebungsvariablen einschränken
 - Nach Möglichkeit darauf verzichten
 - Variablenwerte sind als Input-Daten zu behandeln
 - Längenprüfungen, Prüfungen auf erlaubte Zeichen
- Keine Aufrufe externer Programme über die Shell
 - Programme sind direkt aufzurufen (`execve()`) und nicht über die Shell (`system()` oder `popen()`)
 - Ansonsten sind Angriffe durch Modifikation der Umgebungsvariablen möglich (z.B. PATH)
- Umgebungsunabhängige Schnittstellen verwenden
 - Umgebungsunabhängige Schnittstellen wie `execve()` oder `execle()` benutzen
 - Die Umgebung kann für diese Aufrufe explizit gesetzt werden (Inputparameter im Funktionsaufruf)
 - Andere `exec*`-Schnittstellen (z.B. `execl()`, `execv()`, `execlp()` und `execvp()`) verwenden Systemumgebungsvariablen und sind dadurch angreifbar



Formatstring-Probleme

Formatstrings

- Formatierte Ausgabe von Daten
 - Umwandlungszeichen in einem Formatstring: %d, %x, %p, %s, ...
 - printf() ist eine Funktion mit variabler Parameteranzahl
 - d.h. Parameter werden vom Stack gelesen
- Probleme verursacht durch
 - Verwendung eines Input-Strings als Formatstring
`printf(outPt, inPt);`
 - Direkte Ausgabe eines Strings
`printf (buffer);`
- Relevante Formatierungszeichen
 - Durch Einfügen von Formatierungszeichen in den Inputstring (z.B. %x) können Daten vom Stack **gelesen** werden
 - Durch Einfügen von Formatierungszeichen %n in den Inputstring kann auf den Stack **geschrieben** werden (Länge der bisherigen Ausgabe)
- Konsequenzen:
 - Überschreiben von Rücksprungadresse möglich
 - Ausführung von Exploit-Code möglich

Formatstring - Schutzmaßnahmen

- Sorgfältig programmieren
 - Vermeidung jeglicher Inputstrings als Teil eines Formatstrings bei formatierenden Funktionen:
 - printf(), sprintf(), fprintf(), snprintf()
 - scanf(), sscanf(), fscanf()
 - syslog()
 - Korrekte Nutzung:

```
printf( outPt, "%s", inPt );
```
- Reviews für bestehenden Code durchführen
- Source-Code Analyse Tools verwenden



Tools

Tools

- Unterschiedliche Ansätze:
 - Statisch: nur Source-Code Review
 - Dynamisch: zu Laufzeiten, Source-Code-Modifikationen
- Source-Code Analyse-Tools
 - Prüfen des Source-Codes auf Verwendung potentiell gefährlicher Funktionen, Programm-Fluss kaum berücksichtigt
 - Freie Lösungen: z.B. Flawfinder, ITS4, RATS, Splint, BOON
 - Kommerzielle Produkte: z.B. Fortify, Prexis, CodeAssure, SPI Dynamics
- Laufzeit-Prüftools
 - Erkennung und Vermeidung von Problemen
 - Lösungen: z.B. STOBO, CRED, StackGuard, StackShield, ProPolice SSP, CCured
- Allgemeine Problematik: Verhältnis zwischen korrekt erkannten Problemen und False-Positives

Flawfinder - Beispiel

Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.

Number of dangerous functions in C/C++ ruleset: 158

Examining xxx.c

xxx.c:31: [2] (misc) fopen:

Check when opening files - can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents?.

xxx.c:600: [2] (buffer) memcpy:

Does not check for buffer overflows when copying to destination. Make sure destination can always hold the source data.

xxx.c:742: [2] (buffer) char:

Statically-sized arrays can be overflowed. Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

xxx.c:4597: [4] (buffer) strcpy:

Does not check for buffer overflows when copying to destination. Consider using strncpy or strlcpy (warning, strncpy is easily misused).

Flawfinder - Beispiel (Forts.)

xxx.c:747: [4] (format) vsprintf:

Potential format string problem. Make format string constant.

xxx.c:1095: [1] (buffer) strlen:

Does not handle strings that are not \0-terminated (it could cause a crash if unprotected).

Hits = 64

Lines analyzed = 7713 in 1.34 seconds (9171 lines/second)

Physical Source Lines of Code (SLOC) = 7570

Hits@level = [0] 0 [1] 29 [2] 20 [3] 4 [4] 11 [5] 0

Hits@level+ = [0+] 64 [1+] 64 [2+] 35 [3+] 15 [4+] 11 [5+] 0

Hits/KSLOC@level+ = [0+] 8.45443 [1+] 8.45443 [2+] 4.62351 [3+] 1.98151 [4+] 1.4531 [5+] 0

Minimum risk level = 1

Not every hit is necessarily a security vulnerability.

There may be other security vulnerabilities; review your code!

RATS - Beispiel

Entries in perl database: 33

Entries in python database: 62

Entries in c database: 334

Entries in php database: 55

Analyzing xxx.c

xxx.c:742: High: fixed size local buffer

Extra care should be taken to ensure that character arrays that are allocated on the stack are used safely. They are prime targets for buffer overflow attacks.

xxx.c:747: High: vsprintf

Check to be sure that the format string passed as argument 2 to this function call does not come from an untrusted source that could have added formatting characters that the code is not prepared to handle. Additionally, the format string could contain '%s' without precision that could result in a buffer overflow.

xxx.c:836: High: LoadLibrary

LoadLibrary will search several places for a library if no path is specified, allowing trojan DLL's to be inserted elsewhere even if the intended DLL is correctly protected from overwriting. Make sure to specify the full path.

RATS - Beispiel (Forts.)

xxx.c:4364: High: getenv

Environment variables are highly untrustable input. They may be of any length, and contain any data. Do not make any assumptions regarding content or length. If at all possible avoid using them, and if it is necessary, sanitize them and truncate them to a reasonable length.

xxx.c:6959: High: strcpy

Check to be sure that argument 2 passed to this function call will not copy more data than can be handled, resulting in a buffer overflow.

xxx.c:3951: Low: memcpy

Double check that your buffer is as big as you specify.

When using functions that accept a number n of bytes to copy, such as strncpy, be aware that if the dest buffer size = n it may not NULL-terminate the string.

xxx.c:6955: Low: strlen

This function does not properly handle non-NULL terminated strings. This does not result in exploitable code, but can lead to access violations.

Total lines analyzed: 7677

Total time 0.047000 seconds

163340 lines per second



Best Practices

C-Security Best Practices

- Berücksichtigung aller Input-Parameter
 - Input-Validierung (Länge, gültige Zeichen)
- Verwendung sicherer Konstrukte
 - Kopier- und generell Speicheroperationen
 - Formatstrings
- Ausreichende Dimensionierung von Daten-Buffern
- Vollständige Spezifikation des Verhaltens in Fehlersituationen
- Berücksichtigung aller Abhängigkeiten
 - Programmbibliotheken, Betriebssystem, Umgebung
- Vermeidung von Funktionen, die von Umgebungsvariablen abhängen
- Analyse-Tools in den Entwicklungsprozess integrieren
 - Wahl eines passenden Tools u.U. schwierig

3	4	5	6	7	8
0	0	0	0	0	0
3	4	5	6	7	8
		1			
		2			
2	3	3	3	3	3
-		4			
		5			
6	6	6	6	6	6
		7			
		8			
9	9	9	9	9	9
3	4	5	6	7	8
0	0	0	0	0	0
3	4	5	6	7	8
		1			
		2			
2	3	3	3	3	3
-		4			
		5			
6	6	6	6	6	6
		7			
		8			

Anhang

Warum dieser Vortrag von uns?

- Unsere Erfahrung:

- mehrere Personenjahre in Forschungsprojekten zur sicheren Softwareentwicklung; derzeit gemeinsam mit Partnern wie SAP, Commerzbank, Universitäten, ...
- zahlreiche Schwachstellenanalysen für Softwarehersteller, nebst intensiver Feedbackzyklen mit den Entwicklern
- Erstellung von Anforderungs- und Designspezifikationen in mehreren großen Entwicklungsprojekten
- Erstellung von Guidelines zur sicheren Softwareentwicklung, mit Schwerpunkten Banking & Finance, sowie Webapplikationen
- Reverse Engineering und Gutachten von Sicherheitsfunktionen und Kryptomechanismen
- Implementierung von Sicherheitsfunktionen im Auftrag

Abschlussbemerkung

- die vorliegende Dokumentation wurde von EUROSEC erstellt im Rahmen des secologic Forschungsprojekts, Laufzeit 2005 und 2006, nähere Informationen unter www.secologic.org
- wir bedanken uns beim Bundesministerium für Wirtschaft für die Förderung dieses Projektes
- Anregungen und Feedback sind jederzeit willkommen, ebenso Anfragen zu Sicherheitsaspekten, die hier nicht behandelt werden konnten.

Copyright Hinweis

- Diese Folien wurden von EUROSEC erstellt und dienen der Durchführung von Schulungen oder Seminaren zum Thema Sichere Anwendungsentwicklung, mit Fokus Webapplikationen.
- Wir haben diese Folien veröffentlicht, um die Entwicklung besserer Softwareprodukte zu unterstützen.
- Die Folien dürfen gerne von Ihnen für eigene Zwecke im eigenen Unternehmen verwendet werden, unter Beibehaltung eines Herkunftshinweises auf EUROSEC.
- Eine kommerzielle Verwertung, insbesondere durch Schulungs- oder Beratungsunternehmen, wie beispielsweise Verkauf an Dritte oder ähnliches ist jedoch nicht gestattet.