Speicher-Überschreiber

(*“Buffer overflow”*)

Klaus Kusche

Um was geht es? (1)

In der Beschreibung vieler Sicherheits-Fixes oder CVE's:

"... erlaubt potentiell die Ausführung von beliebigem Code!"

==> Wie???

Um was geht es? (2)

Ursache sind meist ***"Speicherüberschreiber"***.
In diesem Fall ist *"beliebiger Code"*:

- *Echter* ***Maschinencode*** (= Binärcode, *kein* Skript o.ä.)
- Oft nur ***kleine Code-Fragmente***
  (*Kein komplettes Programm*, kein **.exe**-File)
- Oft ***als normale Daten "eingeschmuggelt"***
  (Text, Bild, Video, ...)
- Ausführung nicht als eigens gestarteter Prozess,
  sondern ***innerhalb des laufenden Programmes***
  durch ***Manipulation des Programm-Ablaufes***

Um was geht es? (3)

Code ist oft *sehr kurz*, kann nur *wenig* tun

==> Ist in vielen Fällen *nur erster Schritt* des Exploits, startet eine ganze Kette von Schad-Abläufen:

- ***Shell-Befehl*** oder Shell-Skript starten
- Komplette ***Programme*** aus dem Netz ***nachladen und starten***
- ***Andere Sicherheitslücken*** ausnutzen (z.B. zur *Rechte-Erweiterung*)

==> Uns interessiert hier nur dieser erste Schritt!

Grundidee (1)

***Zwei* Schritte**
zur Entwicklung eines solchen Exploits:

- ***Wie manipuliert man den Programm-Ablauf?***
  (d.h. wie bringt man das Programm dazu, *anderen* Code auszuführen?)

  ==> Dafür muss man eine *bestimmte Art von Fehler im bestehenden Programm* finden und nutzen!

- ***Welches Codefragment führt man dann aus?***

Grundidee (2)

Wenn *Programmfehler* im Zielprogramm *gefunden*
und auszuführender *Code zusammengebastelt* sind:

Sorgfältig ***präparierten Input***
für das Zielprogramm ***konstruieren***

(statt *"ganz normalem Input"*
z.B. via Netz, via Datei, als Texteingabe, ...
heute oft als Bild- oder Video-Datei),

bei ***dessen Verarbeitung*** der ***Fehler auftritt***
und den Schadcode ausführt

Programmablauf manipulieren (1)

Der *eigentliche Code* eines Programms im RAM
ist während der Ausführung meist *schreibgeschützt*
(daher *nicht* manipulierbar)

Aber:

Es gibt ***Pointer auf Code***
im ***normalen Datenbereich***
(*schreibbar* und *benachbart zu "normalen" Daten*),
die den ***Ablauf des Programms steuern***

==> Ein *Überschreiben* dieser Codepointer
lässt das Programm *woanders hinspringen*!

Angriffsziele

- Primär: ***Return-Adressen***
  von Funktionsaufrufen am Stack
- ***Funktionspointer*** und Methodenpointer
- Die ***GOT*** (Global Offset Table)
  und die ***PLT*** (Procedure Linkage Table)
  des *dynamischen Linkers*
- Die Datenstruktur von **`setjmp`** / **`longjmp`**
- In C++: Die ***Typ-Information*** in Objekten

  (die Typ-Information führt zur **`vtable`**,
  die **`vtable`** enthält Pointer auf die virtuellen Methoden)

Programmablauf manipulieren (2)

Diese Codepointer sind zwar

für den Programmierer *nicht direkt* sichtbar,

aber können *überschrieben* werden, wenn man auf

benachbarte "normale" Variablen oder Arrays

zugreift und dabei

über deren reservierten Speicherplatz hinaus schreibt

(z.B. bei einem *Array hinter dessen Ende schreibt*)

Programmablauf manipulieren (3)

Daher der Name ***"Speicherüberschreiber"***:

Es gelingt (durch einen Programmfehler),
Speicherbereiche zu überschreiben,
die eigentlich gar
nicht überschrieben werden sollten / dürften.

Analog ***"Buffer Overflow"***:

Ein Puffer (= Array) ist übergelaufen,
d.h. wurde über sein Ende hinaus beschrieben.

(wobei Arrays nur ein Teil der möglichen Überschreiber sind)

Betroffene Sprachen

Vor allem ***C, C++, Assembler***
(primär Sprachen *ohne* Prüfung
von Pointern & Array-Indices)

Indirekt ***fast alle Sprachen***

… weil die *"Basisfunktionen"* der Libraries
meist in C/C++ geschrieben sind
(besonders: Media-Codec's usw.)

… weil die *Interpreter*, *Garbage-Kollektoren* etc.
oft in C/C++ geschrieben sind

Nutzbare Lücken (1)

- ***Ungeprüfte Input-Länge*** bzw. Input-Größe, vor allem bei ***Strings***
- Meist in Kombination mit ***Arrays fixer Größe***, aber z.B. auch bei zu klein angelegten ***dynamischen Daten***
- ***Array-Grenzen + Pointer-Arithmetik***
- ***"Off-by-one"*-Fehler** in Schleifen, Größenberechnungen, ...
- Programmierfehler bei *Strings*: ***Fehlendes \0***

Nutzbare Lücken (2)

- ***Integer Overflow*** (bei Array-Indices und bei **`malloc`**-Größenberechnungen):

  - *Array-Index-Berechnung* wird durch Überlauf *negativ*:
    Rutscht bei der Index-Prüfung meist durch,
    weil das **`if`** nur auf *"zu groß"* und nicht auf **`<0`** prüft

    ==› Zugriff überschreibt Daten *vor* dem Array

  - **`malloc`**-*Größe* aus **`a*b`** wird durch Überlauf *knapp positiv*:

    - **`malloc`** legt nur einen *ganz kleinen Block* an (erfolgreich!)
    - aber nach erfolgreicher Prüfung auf **`i<a`** und **`j<b`** erlaubt ein Index **`i*j`** beliebigen *Zugriff auf einen Großteil des Speichers*

Nutzbare Lücken (3)

- ***Use-after-free***, ***double-free***, andere **`free`**-Fehler
- **`return`** eines *Pointers auf lokale Daten*, …
- ***Nicht initialisierte*** Variablen,
  vor allem *uninitialisierte Pointer*!
- Funktionen mit *variabel vielen Parametern*
  (in C: Nicht typgeprüft)

Nutzbare Lücken (4)

- Variable **`printf`**-***Format-Strings,***
  oder Argumente passen nicht zum Format
- *Indirekt:*

  Verwendung ***bekannt unsicherer Funktionen***:

  - **`gets`** & **`scanf`** (!!!)
  - **`strcpy`**, **`strncpy`**, **`strcat`**, …
  - **`sprintf`**
  - …

Finden ausnutzbarer Lücken

- Durch ***Lesen des Quellcodes***
  (Open Source oder aus *Reverse Engineering*)
  - … an ***Absturz-Stellen*** (Fuzzer & Debugger)
  - … an Stellen,
    die ***durch Security-Fixes geändert*** wurden
- Nach einem Absturz:
  Durch ***Brute-Force-Versuche***
  mit vielen leicht geänderten Input-Daten

  "Wann tut sich mehr als nur ein Absturz?"

Faustregel

Es ist bei weitem nicht jeder Absturz ausnutzbar
(z.B. explizite **NULL**-Pointer-Zugriffe:
Garantierter *Absturz*, aber im Normalfall *nicht* ausnutzbar),
aber

jeder "ungeklärte" Absturz eines Programmes

wegen eines illegalen Speicherzugriffes

sollte als potentielle Sicherheitslücke

betrachtet und analysiert werden!

Der “Klassiker” (1)

Lokale String-Variable (am Stack)
ohne Längen-Prüfung

==> Füttern mit ***zu langem Eingabe-“String”***,

- der *ausführbaren* ***Maschinencode*** *enthält*
- weiteren Speicher *hinter* dem String *überschreibt*, u.a. die ***Return-Adresse*** *der aktuellen Funktion*
- dort einen ***Pointer*** *auf den eigenen String* ablegt

==> das Return der gerade laufenden Funktion springt dann zum Code im String!!!

Der “Klassiker” (2)

==> Die konstruierten “Angriffsdaten”
haben zwei Funktionen:

1) ***Lösen einen Programmfehler aus,***
der einen ***Codepointer*** gezielt ***überschreibt***

2) ***Enthalten das Code-Fragment***, das dann
über diesen Codepointer angesprungen wird

Für den klassischen *Stack-String-Angriff* muss man
zur Konstruktion des Angriffs-Strings herausfinden:

- *Absolute Adresse* des überschriebenen Strings (zum Anspringen)
- *Abstand* vom String bis zur *Return-Adresse* (zum Überschreiben)

Stand heute

Ständiges "Wettrüsten" zwischen

- immer *besseren Gegenmaßnahmen*
  von Betriebssystem, Compiler, Hardware, ...

  ==> verhindern "triviale" Angriffe

- immer *trickreicheren, komplexeren Angriffen*

  (zB. Manipulation zufällig benachbarter Objekte am Heap, ...)

Z.B. Punkt 2. ***"Einschmuggeln von Code"****:*
Heute oft durch ***Execute Protection verhindert!***

==> Gegen-Trick "Return Oriented Programming"

“Return Oriented Programming”

Stack so manipulieren,

- dass die ***Return-Adresse***
  auf ein ***bestehendes Codestück*** zeigt
  (im angegriffenen Programm,
  aber vor allem *in Libraries*, z.B. “**`system`**”)
- und dass eventuell rundherum am Stack
  die “richtigen” ***Parameter & lokalen Variablen***
  für dieses Codestück stehen
  (z.B. eine **`system`**-Befehlszeile
  mit “bösen” Befehlen)

Auch Lesen kann gefährlich sein...

Beispiel: ***Heartbleed-Bug*** in OpenSSL

Ursache: ***Fehlende Konsistenzprüfung*** zwischen
- im Paket *explizit angegebener Datenmenge*
- *tatsächlicher Größe* des Datenpaketes

Effekt:

Nur *x Bytes* Daten *gespeichert*,
trotzdem *y Bytes ausgelesen*
und über's Netz *zurückgeschickt*

==> y-x Bytes interne Speicherinhalte geleakt,
potentiell hoch sensible Daten!

Gegenmaßnahmen (0)

"Sichere" Programmiersprachen verwenden:

- *Index-Prüfung* aller *Array-Zugriffe*

  (in C formal gar nicht lückenlos möglich!)
- *Keine* für den Programmierer sichtbaren *Pointer*, *keine Pointer-Arithmetik*
- Keine explizite dynamische Speicherverwaltung (kein **`free`** / **`delete`**), *nur Garbage Collection*

Gegenmaßnahmen (1)

Compiler-Option "Stack Smashing Protector":

Speicherüberschreiber am Stack
arbeiten sehr oft *sequentiell von unten nach oben*

==› Wert *unmittelbar unterhalb* der Return-Adresse
wird auch *meist überschrieben*!

- Dort einen zusätzlichen Speicherplatz reservieren
  und am Beginn jedes Funktionsaufrufs
  einen *nicht vorhersagbaren Wert* speichern
- Unmittelbar vor dem Return prüfen,
  ob dieser *Wert noch unversehrt* ist (sonst Programm-Abbruch)

==› *Schützt nur vor Stack-Angriffen auf die Return-Adresse*

Gegenmaßnahmen (2)

Compiler-Option "vtable Verification":

- Compiler erstellt *zusätzliche Tabelle* für jede *Klasse*:
  Alle *gültigen* **`vtable`**-Pointer
  für *diese und davon abgeleitete* Klassen
- … und *zusätzlichen Code vor jedem virtuellen Call*:
  Prüft, ob tatsächlicher **`vtable`**-*Pointer im Objekt*
  zu diesem Call *passt* (in der Tabelle ist)

==> verhindert virtuelle Calls mit
manipuliertem **`vtable`**-*Pointer im Objekt*
auf eine "getürkte" Vtable (die auf Schadcode zeigt)

Gegenmaßnahmen (3)

Intel "CET" *("Control Flow Enforcement Technology")*

"Schatten-Stack" in *Hardware*,
nicht direkt manipulierbar (aber leider endlich)

- ... speichert automatisch bei jedem Call-Befehl
  die *Return-Adresse*
- ... *vergleicht* bei jedem Return
  die Return-Adresse am Stack
  mit der am "Schatten-Stack"
- ... *unterbricht* das Programm bei Abweichungen

==› Schützt auch *nur Return-Adressen*, und das nicht lückenlos...

Gegenmaßnahmen (4)

Execute Protection
("NX-Bit", "DEP", "W^X", ...):

Derselbe Speicherbereich darf
nie schreibbar und ausführbar zugleich
sein!

(Code-Bereiche sind nicht schreibbar,
Daten-Bereiche sind nicht ausführbar)

==> Ausführung von eingeschleustem Code in Daten wird *verhindert*! (und das sogar "*kostenlos*"!)

Gegenmaßnahmen (5)

Realisierung der Execute Protection über

Zugriffsrechte-Bits in den *Pagetables*

==> Benötigt im Regelfall

- ***Prozessor mit MMU*** (Memory Management Unit)
- Betriebssystem mit ***virt. Speicherverwaltung***

... oder zumindest Prozessor mit MPU (Memory Protection Unit)

==> **Keine Execute Protection**
auf *"ganz kleinen"* Prozessoren (z.B. Embedded Systems)

==> Hilft *nicht* gegen *"Return Oriented Programming"*,
aber gegen jeden als Daten *eingeschleusten Code*

Gegenmaßnahmen (6)

Problem der Execute-Protection:

Sehr viele Programme enthalten heute JIT's
(***"Just in Time Compiler"***):

- Fast alle Skriptsprachen-Interpreter,
  damit auch *alle Browser, Mail-Programme usw.*
- Pattern- und Regular-Expression-Matching
- Manche Grafik-Treiber

JIT muss *zur Laufzeit Code schreiben & ausführen*

==> Programme mit JIT müssen zumindest teilweise
***ohne* Execute-Protection ausgeführt** werden

Gegenmaßnahmen (7)

ASLR "Address Space Layout Randomization":

Der Loader des Betriebssystems variiert
bei jedem Programmstart zufällig die *Adressen*

- des **Programms** (incl. globaler Daten & Heap)
  ("PIE" = "Position Independent Executable")
- aller **Shared Libraries**
- des **Stacks**, dynamischer **Datenbereiche**, **`mmap`**'s

==> *Fix codierte Adressen in Exploits für Code & Daten gehen ins Leere*

==> *Kein absoluter Schutz, aber mehr Hack-Aufwand bzw. kleinere Treffer-Wahrscheinlichkeit*

Gegenmaßnahmen (8)

Sandboxing:

Filterung aller System Calls eines Programms
(z.B. keine Programmstarts,
keine oder nur eingeschränkte Datei-Zugriffe, ...)

Primitive Urform unter Unix/Linux: **`chroot`**, Linux **`seccomp`**

Heute *diverse Sicherheits-Erweiterungen*
- ***RBAC*** (Role based access control)
- ***Capabilities*** (SELinux, AppArmor, ...)
- **`seccomp-bpf`**

==> *Hilft nicht* gegen Überschreiber,
aber mindert deren mögliche Folgen

Gegenmaßnahmen (9)

Brute-Force-Bremse:

Das Suchen der richtigen *Exploit-Adresse*
durch *Ausprobieren* vieler Inputs
verursacht ev. *viele Programm-Abstürze*

==> ***Automatischen Neustart***
(z.B. von Server-Diensten)
- nach Abstürzen ***verzögern***,
- nach mehreren Abstürzen ganz ***unterbinden***

==> *Verhindert Brute-Force-Ausprobieren*
von Ziel-Adressen, Lage von Variablen, ...

Gegenmaßnahmen (10)

Trusted Path Execution:

- Fixe ***Liste von Programm-Directories***
  Diese Directories sind für normale User bzw. normale Programme ***nicht schreibbar***!
- Programme dürfen ***nur gestartet*** werden, wenn sie ***in einem dieser Directories*** liegen, nicht mit beliebigem relativen/absoluten Pfad

==> Hilft *nicht* gegen *Speicherüberschreiber*, aber ev. gegen den 2. Schritt eines Exploits, nämlich das *Ausführen heruntergeladener Dateien*

Fehler-Vermeidung (1)

- Programmier-***Richtlinien,***
  Schulungen
- ***Textuelle Suche***
  nach gefährlichen Funktionen usw.
- ***Compiler-Warnungen*** aufdrehen (!)
  (mit *maximaler Optimierung* zwecks *Datenfluß-Analyse*)
- ***Statische Programmanalyse*** *(“Lint”)* einsetzen
- ***Code Reviews***

Fehler-Vermeidung (2)

Fortgeschrittene Tools für
statische Programm-Analyse (lesen den Quellcode)

... analysieren für jede ***Variable***
& jede Stelle im Programm
den ***möglichen Wertebereich***
(auch: *"Ist noch uninitialisiert"*)

... merken sich z.B.
- ob für eine Pointer-Variable
schon **`free`** / **`delete`** aufgerufen wurde
- ob mehrere Pointer *auf dasselbe zeigen*

Fehler-Vermeidung (3)

Fortgeschrittene statische Programm-Analyse

... findet einige ***fehlende*** **`if`**`'s`, ***Off-by-one-Fehler***, ...
(z.B. Werte-Bereich einer Index-Variable
ist größer als Index-Bereich des Arrays,
Pointer könnte **`NULL`** *sein und wird nicht geprüft, ...)*

... findet einige ***use-after-free***, ***double-free*** usw.

Aber leider:

- Findet bei weitem nicht alles
- Oft *sehr viele "false positives"!*

Fehler-Vermeidung (4)

Speicherzugriffs-Checker verwenden:

- *Compiler*-basiert, z.B. ***ASAN***
  (wichtigstes Google-Tool, im **`gcc`** und **`llvm`**)
- Code *Instrumenter*, z.B. ***Purify***
- *Interpretierende* Tools, z.B. ***Valgrind***

... prüfen alle *Array- und Pointer-Zugriffe*

... führen Buch über *allokierte / freigegebene Blöcke*

... führen Buch über *uninitialisierte Speicherbereiche*

Fehler-Vermeidung (5)

Speicherzugriffs-Checker

... finden die Fehler der Fehler-Arten,
die sie finden können/sollen, ***praktisch zu 100 %***

... erfordern *Verständnis des Testers*,
welche Fehler-Arten das Tool erkennt
(und welche nicht!)

... aber sind ***für Produktivcode viel zu langsam!***

==> *Nur **in der Entwicklung zum Testen** verwenden,*
für Produktiv-Einsatz leider nicht brauchbar!

Fehler-Vermeidung (6)

Testen!!!

- "Kreative" ***menschliche Tester***,
  spezielle ***Pen-Tester***,
  Wettbewerbe & Bug Bounties
- ***Fuzzing-Programme***

  ... füttern ein Programm tausende Male
  automatisch mit zufällig generiertem Input

  Fortgeschrittene Variante:
  Analysieren interne Reaktion des Programms
  und *modifizieren Input gezielt*