Speicher-Überschreiber

Klaus Kusche

Allgemeines (1)

Über drei Viertel aller Sicherheitslücken
*entfallen auf wenige **"Standard-Fehlerarten"**!*

***"Echt falscher"** Code* (= *macht das Falsche*) ist *selten*!
Stattdessen meist:

Sicherheitslücke = Fehlender Code

- *Fehlende Input-Prüfungen*
- Fehlende Returncode-Prüfungen, ...

oder auch: Verwendung *unsicherer Konstrukte*

==> "Schlamperei" & "Fehlendes Wissen"

Allgemeines (2)

Jeder kann jeden Code
auf Sicherheitslücken untersuchen!

- Durch ***Reverse Engineering*** = Disassemblierung
- Durch ***gezielte Tests*** mit konstruiertem Input

==> *Irgendwann werden die Lücken gefunden!*
(u.a. wegen finanziellem Anreiz)

Nachträgliche Behebung ist meist
viel teurer als *"gleich richtig machen"*
(finanziell, vom Ruf her, ...)

Grundidee

Bestehendes Programm füttern mit

sorgfältig ***präpariertem Input***
(via Netz oder via Datei,
heute z.B. auch via Bild- oder Video-Datei)

==> Ausführen von (*beliebigem* eigenem) Code

durch *"unplanmäßige" Manipulation*
von *Speicherinhalten*,
die den *Programmablauf beeinflussen*

Der “Klassiker”

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 Code* ablegt

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

Betroffene Sprachen

Primär nur ***C, C++, Assembler***
(nur 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)

- ***Array-Grenzen + Pointer-Arithmetik***
  (primär am Stack & am Heap)
- ***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 (2)

- Programmierfehler bei Strings: ***Fehlendes*** **`\0`**
- ***“Off-by-one”***-Fehler
- ***Use-after-free***, ***double-free***,
  return eines Pointers auf lokale Daten, …
- ***Nicht initialisierte*** Variablen
- Variable **`printf`**-***Format-Strings***!

Nutzbare Lücken (3)

- *Indirekt:*

  Verwendung ***bekannt unsicherer Funktionen***:

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

Angriffsziele

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

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

Klassischer Angriffsweg

- ***Eigenen Binär-Code*** in irgendwelchen *Daten* (String, Bild, …) einbauen
- Diesen Code *anspringen durch*
  - Manipulation einer ***Return-Adresse*** am *Stack*
  - Manipulation eines ***Function Pointers*** o.ä.

Heute meist durch Execute Protection usw. verhindert!

“Return Oriented Programming”

Stack so manipulieren,

- dass die ***Return-Adresse***
  auf eine ***bestehende Funktion*** zeigt
  (im Programm,
  aber vor allem *in Libraries*, z.B. “**`system`**”).
- und dass darüber am Stack ***Parameter***
  für diesen Funktionsaufruf stehen
  (z.B. eine **`system`**-Befehlszeile
  mit “bösen” Befehlen)

Entwicklung von Exploits

- 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?"

Gegenmaßnahmen (0)

"Sichere" Programmiersprachen verwenden:

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

  (in C gar nicht möglich)
- *Keine* für den Programmierer sichtbaren *Pointer*, *keine Pointer-Aritmetik*
- Keine explizite dynamische Speicherverwaltung, *nur Garbage Collection*

Gegenmaßnahmen (1)

Compiler-Option "Stack Protector":

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

==> Wert *unmittelbar unter* der Return-Adresse
wird auch *meist überschrieben*!

- Dort beim Funktionsaufruf
  einen *nicht vorhersagbaren Wert* speichern
- Vor dem Return prüfen, ob *Wert unversehrt* ist
  (sonst Programm-Abbruch)

Gegenmaßnahmen (2)

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

"Schatten-Stack" in *Hardware*,
nicht direkt manipulierbar

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

Gegenmaßnahmen (3)

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

Über *Zugriffsrechte-Bits*
in den Pagetables der ***MMU***:

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*!

Gegenmaßnahmen (4)

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
***ohne* Execute-Protection ausgeführt** werden

Gegenmaßnahmen (5)

ASLR "Address Space Layout Randomization":

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

- des **Programms** (incl. globaler Daten)
  ("PIE" = "Position Independent Executable")
- aller **Shared Libraries**
- des **Stacks**
- dynamisch angeforderter **Datenbereiche** (**`mmap`**)

==> *Fix codierte Adressen, Daten-Abstände usw.*
im Exploit gehen ins Leere

Gegenmaßnahmen (6)

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`**

Gegenmaßnahmen (7)

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-Exploits*

Gegenmaßnahmen (8)

Trusted Path Execution:

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

- 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

Fehler-Vermeidung (1)

- Programmier-***Richtlinien***
- ***Textuelle Suche***
  nach gefährlichen Funktionen usw.
- ***Compiler-Warnungen*** aufdrehen (!)
- ***"Lint"*** einsetzen
- ***Schulungen***
- ***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)

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

Aber leider:

- Findet 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***
- ***Fuzzing-Programme***

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

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