Lær at lave shellcode

br0ns iDolf Hatler TM
Datalogisk institut, Københavns universitet

Disclaimer

SPØRG FOR FANDEN HVIS DET GÅR FOR STÆRKT!

Hvad er målet?

Hvad er målet?

- Kontrol over datamaten.

Hvad er målet?

- Kontrol over datamaten.
- Eksekvering af vores kode.

Hvad er målet?

- Kontrol over datamaten.
- Eksekvering af vores kode.
- Shell.

exploitable.c

```
 1 #include <string.h>
 2 #include <stdio.h>
 3 #define SIZE 32
 4
 5 void copy_stuff(char *to, char *from) {
 6     while(*from) {
 7         *to = *from;
 8         from++;to++;
 9     }
10 }
11
12 int main(int argc, char **argv) {
13     char buffer[SIZE];
14     memset(buffer, 0, SIZE);
15
16     copy_stuff(buffer, argv[1]);
17
18     printf("%s\n", buffer);
19 }
```

Demo

Hvad nu?

Hvad nu?

- Den hoppede til adresse `0x41414141`.

Hvad nu?

- Den hoppede til adresse `0x41414141`.
- Kan vi bruge det?

Hvad nu?

- Den hoppede til adresse `0x41414141`.
- Kan vi bruge det?
- Hvordan kommer vi videre?

Lynkursus i maskinarkitektur

Lynkursus i maskinarkitektur

- Hardware

Lynkursus i maskinarkitektur

- Hardware
- Instruktionssættet

Lynkursus i maskinarkitektur

- Hardware
- Instruktionssættet
- Styresystemet

Hardware

Hardware

- CPU

Hardware

- CPU
- Hukommelse

Hardware

- CPU
- Hukommelse
- IO

CPU

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)
- Har en instruktionspeger (`EIP`)

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)
- Har en instruktionspeger (`EIP`)
- Kan regne og gøre mange smarte ting (cache, hukommelsesmaskering, etc.)

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)
- Har en instruktionspeger (`EIP`)
- Kan regne og gøre mange smarte ting (cache, hukommelsesmaskering, etc.)

Algoritme:

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)
- Har en instruktionspeger (`EIP`)
- Kan regne og gøre mange smarte ting (cache, hukommelsesmaskering, etc.)

Algoritme:

1. Læs en instruktion fra hukommelsen og flyt `EIP`

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)
- Har en instruktionspeger (`EIP`)
- Kan regne og gøre mange smarte ting (cache, hukommelsesmaskering, etc.)

Algoritme:

1. Læs en instruktion fra hukommelsen og flyt `EIP`
2. Gør hvad instruktionen siger

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)
- Har en instruktionspeger (`EIP`)
- Kan regne og gøre mange smarte ting (cache, hukommelsesmaskering, etc.)

Algoritme:

1. Læs en instruktion fra hukommelsen og flyt `EIP`
2. Gør hvad instruktionen siger
3. Gå til punkt 1

CPU

- Har registre der bruges til midlertidige beregninger (`EAX`, `EBX`, `EBP`, `ESP`, `...`)
- Har en instruktionspeger (`EIP`)
- Kan regne og gøre mange smarte ting (cache, hukommelsesmaskering, etc.)

Algoritme:

1. Læs en instruktion fra hukommelsen og flyt `EIP`
2. Gør hvad instruktionen siger
3. Gå til punkt 1

Eksempler: `mov`, `add`, `jmp`, `call`, `push`

Og en pony

Hukommelse

Hukommelse

Hukommelse på X86 er kompliceret.

Hukommelse

Hukommelse på X86 er kompliceret.

Long story short:

Hukommelse

Hukommelse på X86 er kompliceret.

Long story short:

- Fra jeres synsvinkel består hukommelsen af adresser fra `0x00000000` til `0xffffffff`.

Hukommelse

Hukommelse på X86 er kompliceret.

Long story short:

- Fra jeres synsvinkel består hukommelsen af adresser fra `0x00000000` til `0xffffffff`.
- Hver adresse indeholder 1 byte, men man tilgår normalt 4 bytes af gangen (så `0x1234` er `0x1234-0x1237`).

Hukommelse

Hukommelse på X86 er kompliceret.

Long story short:

- Fra jeres synsvinkel består hukommelsen af adresser fra `0x00000000` til `0xffffffff`.
- Hver adresse indeholder 1 byte, men man tilgår normalt 4 bytes af gangen (så `0x1234` er `0x1234-0x1237`).
- X86 er little endian (den "mindst betydende byte" bliver gemt først). Tallet `0xDECAFBAD` består af de fire bytes `0xAD`, `0xFB`, `0xCA` og `0xDE` *i den rækkefølge*.

Hukommelse

Hukommelse på X86 er kompliceret.

Long story short:

- Fra jeres synsvinkel består hukommelsen af adresser fra `0x00000000` til `0xffffffff`.
- Hver adresse indeholder 1 byte, men man tilgår normalt 4 bytes af gangen (så `0x1234` er `0x1234-0x1237`).
- X86 er little endian (den "mindst betydende byte" bliver gemt først). Tallet `0xDECAFBAD` består af de fire bytes `0xAD`, `0xFB`, `0xCA` og `0xDE` *i den rækkefølge*.
- Hukommelsen er inddelt i forskellige områder. Hvis man prøver at tilgå noget udenfor et af disse områder (*segmenter*) får man en segmenteringsfejl.

Hukommelse

Hukommelse på X86 er kompliceret.

Long story short:

- Fra jeres synsvinkel består hukommelsen af adresser fra `0x00000000` til `0xffffffff`.
- Hver adresse indeholder 1 byte, men man tilgår normalt 4 bytes af gangen (så `0x1234` er `0x1234-0x1237`).
- X86 er little endian (den "mindst betydende byte" bliver gemt først). Tallet `0xDECAFBAD` består af de fire bytes `0xAD`, `0xFB`, `0xCA` og `0xDE` *i den rækkefølge*.
- Hukommelsen er inddelt i forskellige områder. Hvis man prøver at tilgå noget udenfor et af disse områder (*segmenter*) får man en segmenteringsfejl.
- Adresserne er lokale for dit program. Flere processer kan bruge `0x1234` uden at tilgå det samme stykke fysiske hukommelse (som endda kan være swap på en disk).

Hukommelse - fortsat

Relevante områder i hukommelsen er:

Hukommelse - fortsat

Relevante områder i hukommelsen er:

- Programområdet (`EIP`)

Hukommelse - fortsat

Relevante områder i hukommelsen er:

- Programområdet (`EIP`)
- Statisk allokerede variable (BSS)

Hukommelse - fortsat

Relevante områder i hukommelsen er:

- Programområdet (`EIP`)
- Statisk allokerede variable (BSS)
- Dataområdet

Hukommelse - fortsat

Relevante områder i hukommelsen er:

- Programområdet (`EIP`)
- Statisk allokerede variable (BSS)
- Dataområdet
- Hobområdet (dynamisk lager)

Hukommelse - fortsat

Relevante områder i hukommelsen er:

- Programområdet (`EIP`)
- Statisk allokerede variable (BSS)
- Dataområdet
- Hobområdet (dynamisk lager)
- Stakområdet (`ESP`/`EBP`)

IO

IO

- Det findes

IO

- Det findes
- Du er ligeglad

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)
- Hop (`jmp, jne, call, ret`)

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)
- Hop (`jmp, jne, call, ret`)
- Stakoperationer (`push, pop`)

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)
- Hop (`jmp, jne, call, ret`)
- Stakoperationer (`push, pop`)
- `nop`

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)
- Hop (`jmp, jne, call, ret`)
- Stakoperationer (`push, pop`)
- `nop`

Og et par funky:

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)
- Hop (`jmp, jne, call, ret`)
- Stakoperationer (`push, pop`)
- `nop`

Og et par funky:

- `bsf` (Bit Scan Forward)

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)
- Hop (`jmp, jne, call, ret`)
- Stakoperationer (`push, pop`)
- `nop`

Og et par funky:

- `bsf` (Bit Scan Forward)
- `aesenc` ("Perform One Round of an AES Encryption Flow")

Instruktionssættet

Generelt er X86 et clusterfuck, men her er nogle nemme:

- Flyt og regn på data (`mov, add, xor, shl`)
- Hop (`jmp, jne, call, ret`)
- Stakoperationer (`push, pop`)
- `nop`

Og et par funky:

- `bsf` (Bit Scan Forward)
- `aesenc` ("Perform One Round of an AES Encryption Flow")
- `f2xm1` (Udregn $2^x - 1$ i et floating point -register)

Legetime 1

Legetime 1

Mål: kald `/bin/sh`

Legetime 1

Mål: kald `/bin/sh`

C: `execve(char *filename, char *argv[], char *envp[])`

Legetime 1

Mål: kald `/bin/sh`

C: `execve(char *filename, char *argv[], char *envp[])`

Aassembler: Sæt `eax = 11`, `ebx = filename`, `ecx = argv`, `edx = envp`.
Kald derefter på styresystemet med `int 0x80`.

Legetime 1

Mål: kald `/bin/sh`

C: `execve(char *filename, char *argv[], char *envp[])`

Aassembler: Sæt `eax = 11`, `ebx = filename`, `ecx = argv`, `edx = envp`.
Kald derefter på styresystemet med `int 0x80`.

Nyttige instruktioner:

`[BITS 32]`	Skal stå på først linje for få nasm til at bruge 32-bit
`global _start`	Gør entry-point synligt for linkeren
`_start:`	Entry-point i programmet
`mov r1, n`	Flytter tallet n til register `r1`
`mov r1, r2`	Flytter indholdet af register `r2` til register `r1`
`mov r1, label`	Flytter adressen af `label` til `r1`
`int 0x80`	Kald styresystemsfunktioner
`str1: db "pony",0`	Gemmer den nul-terminerede tekst "pony" i hukommelsen med `str1` som peger til den
`array1: dd l1,l2,l3,0`	Gemmer et array af pointers (`l1,l2,l3`) i hukommelsen med `array1` som peger

Legetime 1

Mål: kald `/bin/sh`

C: `execve(char *filename, char *argv[], char *envp[])`

Aassembler: Sæt `eax = 11`, `ebx = filename`, `ecx = argv`, `edx = envp`.
Kald derefter på styresystemet med `int 0x80`.

Nyttige instruktioner:

`[BITS 32]`	Skal stå på først linje for få nasm til at bruge 32-bit
`global _start`	Gør entry-point synligt for linkeren
`_start:`	Entry-point i programmet
`mov r1, n`	Flytter tallet n til register `r1`
`mov r1, r2`	Flytter indholdet af register `r2` til register `r1`
`mov r1, label`	Flytter adressen af `label` til `r1`
`int 0x80`	Kald styresystemsfunktioner
`str1: db "pony",0`	Gemmer den nul-terminerede tekst "pony" i hukommelsen med `str1` som peger til den
`array1: dd l1,l2,l3,0`	Gemmer et array af pointers (`l1,l2,l3`) i hukommelsen med `array1` som peger

Kør med:
`nasm -f elf evil.asm; ld -melf_i386 -o evil evil.o; ./evil`

Demo

Styresystemet

Styresystemet

- Du snakker med styresystemet ved hjælp af `int 0x80`

Styresystemet

- Du snakker med styresystemet ved hjælp af `int 0x80`
- `EAX` er funktionsnummeret (11 = `execve`). Vælg andre tal for andre funktioner[1].

[1] `http://asm.sourceforge.net/syscall.html`

Styresystemet

- Du snakker med styresystemet ved hjælp af `int 0x80`
- `EAX` er funktionsnummeret (11 = `execve`). Vælg andre tal for andre funktioner[1].
- `EBX`, `ECX`, `EDX`, `ESI`, `EDI` er argumenter (i den rækkefølge).

[1] `http://asm.sourceforge.net/syscall.html`

Og en pony mere

Back on track 1

Back on track 1

- Vi har nu oversat vores shellcode som et selvstændigt program, der kan køres alene.

Back on track 1

- Vi har nu oversat vores shellcode som et selvstændigt program, der kan køres alene.
- Problem: Vi bruger labels.

Back on track 1

- Vi har nu oversat vores shellcode som et selvstændigt program, der kan køres alene.
- Problem: Vi bruger labels.
- Man kan ikke bruge (den her slags) labels i shellcode.

Back on track 1

- Vi har nu oversat vores shellcode som et selvstændigt program, der kan køres alene.
- Problem: Vi bruger labels.
- Man kan ikke bruge (den her slags) labels i shellcode.
- Men hvordan får vi fat i vores data uden labels?

Et muligt svar: Brug stakken

Et muligt svar: Brug stakken

Idé: Du har en peger til stakken (`ESP`) og medmindre der er gået noget galt har du lov til at gemme data der. Det gør du ved at `push`'e:

Et muligt svar: Brug stakken

Idé: Du har en peger til stakken (`ESP`) og medmindre der er gået noget galt har du lov til at gemme data der. Det gør du ved at `push`'e:

- Afhænger af størrelsen, indsæt selv joke

Et muligt svar: Brug stakken

Idé: Du har en peger til stakken (`ESP`) og medmindre der er gået noget galt har du lov til at gemme data der. Det gør du ved at `push`'e:

- Afhænger af størrelsen, indsæt selv joke
- `push byte 0x68 ; pusher 0x00000068`

Et muligt svar: Brug stakken

Idé: Du har en peger til stakken (`ESP`) og medmindre der er gået noget galt har du lov til at gemme data der. Det gør du ved at `push`'e:

- Afhænger af størrelsen, indsæt selv joke
- `push byte 0x68 ; pusher 0x00000068`
- `push word 0x732f ; pusher 0x732f`

Et muligt svar: Brug stakken

Idé: Du har en peger til stakken (`ESP`) og medmindre der er gået noget galt har du lov til at gemme data der. Det gør du ved at `push`'e:

- Afhænger af størrelsen, indsæt selv joke
- `push byte 0x68 ; pusher 0x00000068`
- `push word 0x732f ; pusher 0x732f`
- `push dword 0x6e69622f ; pusher 0x6e69622f`

Et muligt svar: Brug stakken

Idé: Du har en peger til stakken (`ESP`) og medmindre der er gået noget galt har du lov til at gemme data der. Det gør du ved at `push`'e:

- Afhænger af størrelsen, indsæt selv joke
- `push byte 0x68 ; pusher 0x00000068`
- `push word 0x732f ; pusher 0x732f`
- `push dword 0x6e69622f ; pusher 0x6e69622f`

Kortere:

- `push 0x68732f6e`
- `push 0x69622f2f`

Et muligt svar: Brug stakken

Idé: Du har en peger til stakken (`ESP`) og medmindre der er gået noget galt har du lov til at gemme data der. Det gør du ved at `push`'e:

- Afhænger af størrelsen, indsæt selv joke
- `push byte 0x68 ; pusher 0x00000068`
- `push word 0x732f ; pusher 0x732f`
- `push dword 0x6e69622f ; pusher 0x6e69622f`

Kortere:

- `push 0x68732f6e`
- `push 0x69622f2f`

Og med en smart syntax:

- `push `n/sh``
- `push `//bi``

Se desuden `man ascii`.

Legetime 2

Legetime 2

Mål: Lav din shellcode om, så den ikke har nogen labels.

Legetime 2

Mål: Lav din shellcode om, så den ikke har nogen labels.

Nyttige instruktioner:

`[BITS 32]`	Skal stå på først linje for få nasm til at bruge 32-bit
`push n`	Pusher tallet n til stakken som 4 bytes
`push word n`	Pusher tallet n til stakken som 2 bytes
`push `abcd``	Pusher de 4 bytes `a`, `b`, `c` og `d` til stakken
`mov r1, r2`	Flytter indholdet af register `r2` til register `r1`
`mov r1, esp`	Flytter stakpegeren til register `r1`
`int 0x80`	Udfører systemkald

Legetime 2

Mål: Lav din shellcode om, så den ikke har nogen labels.

Nyttige instruktioner:

`[BITS 32]`	Skal stå på først linje for få nasm til at bruge 32-bit
`push n`	Pusher tallet n til stakken som 4 bytes
`push word n`	Pusher tallet n til stakken som 2 bytes
`push `abcd``	Pusher de 4 bytes `a`, `b`, `c` og `d` til stakken
`mov r1, r2`	Flytter indholdet af register `r2` til register `r1`
`mov r1, esp`	Flytter stakpegeren til register `r1`
`int 0x80`	Udfører systemkald

Kør med:

```
nasm -f bin evil.asm; ../demo evil
```

Demo

Back on track 2

Back on track 2

- Nu har vi shellcode som ikke behøver en bestemt kontekst for at virke.

Back on track 2

- Nu har vi shellcode som ikke behøver en bestemt kontekst for at virke.
- Den kan smides ind i et andet program og gøre hvad vi har lyst.

Back on track 2

- Nu har vi shellcode som ikke behøver en bestemt kontekst for at virke.
- Den kan smides ind i et andet program og gøre hvad vi har lyst.
- Problem: Mange steder hvor vi har mulighed for at lirke shellcode ind kræver at den ikke indeholder NULL-tegn.

Back on track 2

- Nu har vi shellcode som ikke behøver en bestemt kontekst for at virke.
- Den kan smides ind i et andet program og gøre hvad vi har lyst.
- Problem: Mange steder hvor vi har mulighed for at lirke shellcode ind kræver at den ikke indeholder NULL-tegn.
- Hvordan fjerner vi dem?

Trickz til at undgå NULL-bytes

Trickz til at undgå NULL-bytes

Ingen NULL-bytes i koden?

Trickz til at undgå NULL-bytes

Ingen NULL-bytes i koden?

Ofte intet problem, men hvad hvis et register skal være tomt (eller næsten tomt)?

Trickz til at undgå NULL-bytes

Ingen NULL-bytes i koden?

Ofte intet problem, men hvad hvis et register skal være tomt (eller næsten tomt)?

Der findes masser af tricks der kan bruges i forskellige situationer:

Trickz til at undgå NULL-bytes

Ingen NULL-bytes i koden?

Ofte intet problem, men hvad hvis et register skal være tomt (eller næsten tomt)?

Der findes masser af tricks der kan bruges i forskellige situationer:

```
1 xor ebx, ebx   ; virker altid
2 mul ebx        ; hvis ebx nul      -> eax+edx tom
3 shl eax, 31    ; hvis eax lige     -> eax tom
4 shr eax, 31    ; hvis eax positiv -> eax tom
5 cdq            ; hvis eax positiv -> edx tom
6
7 push byte 7
8 pop eax        ; eax = 7
```

Legetime 3

Lav jeres shellcode om til ikke at indeholde NULL-tegn.

Demo

Spørgsmål?