OSY.SSI[2015][10]

Software Lab: Code injection 2

ROP and UAF

ROP and UAF

We will explore two interesting exploitations techniques :

ROP and UAF

We will explore two interesting exploitations techniques :

- return-oriented programming (ROP), first used to defeat code signing and non-executable stacks

ROP and UAF

We will explore two interesting exploitations techniques :

- return-oriented programming (ROP), first used to defeat code signing and non-executable stacks
- use-after-free (UAF), a kind of heap attack that bypasses stack protections altogether

ROP and UAF

We will explore two interesting exploitations techniques :

- return-oriented programming (ROP), first used to defeat code signing and non-executable stacks
- use-after-free (UAF), a kind of heap attack that bypasses stack protections altogether

Both techniques are widely used today.

Return-oriented programming

Question:

Return-oriented programming

Question: how to bypass non-executable stacks?

Return-oriented programming

0. Faking a stack

Task:

Return-oriented programming

0. Faking a stack

Task:

- Read, understand and compile the `showdate.c` program

Return-oriented programming

0. Faking a stack

Task:

- Read, understand and compile the `showdate.c` program
- What does it do? Check with `objdump` that `not_called` is present

Return-oriented programming

0. Faking a stack

Task:

- Read, understand and compile the `showdate.c` program
- What does it do? Check with `objdump` that `not_called` is present
- Use `gdb` to find the addresses we need:

Return-oriented programming

0. Faking a stack

Task:

- Read, understand and compile the `showdate.c` program
- What does it do? Check with `objdump` that `not_called` is present
- Use `gdb` to find the addresses we need:

```
gdb showdate
print 'system@plt'
x/s not_used
```

Return-oriented programming

0. Faking a stack

Task:

- Read, understand and compile the `showdate.c` program
- What does it do? Check with `objdump` that `not_called` is present
- Use `gdb` to find the addresses we need:

```
gdb showdate
print 'system@plt'
x/s not_used
```

- Now assemble a fake stack and send it to `showdate`

Return-oriented programming

0. Faking a stack

Task:

- Read, understand and compile the `showdate.c` program
- What does it do? Check with `objdump` that `not_called` is present
- Use `gdb` to find the addresses we need:

```
gdb showdate
print 'system@plt'
x/s not_used
```

- Now assemble a fake stack and send it to `showdate`

```
AAA...AAA + BBBB + (address of system) + CCCC + (address of not_used)
```

Return-oriented programming

0. Faking a stack

Task:

- Read, understand and compile the `showdate.c` program
- What does it do? Check with `objdump` that `not_called` is present
- Use `gdb` to find the addresses we need:

```
gdb showdate
print 'system@plt'
x/s not_used
```

- Now assemble a fake stack and send it to `showdate`

```
AAA...AAA + BBBB + (address of system) + CCCC + (address of not_used)
```

- What happens on exit? Why? More on that in a minute.

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`
- Break at `main`, then run.

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`
- Break at `main`, then run.
- Now libraries are loaded. Find the address of `system`.

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`
- Break at `main`, then run.
- Now libraries are loaded. Find the address of `system`.
- Use `find X, +999999999, "/bin/sh"` where X is the address of system.

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`
- Break at `main`, then run.
- Now libraries are loaded. Find the address of `system`.
- Use `find X, +999999999, "/bin/sh"` where X is the address of system.
- Write down this address Y ! Now back to `showdate`, send it

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`
- Break at `main`, then run.
- Now libraries are loaded. Find the address of `system`.
- Use `find X, +999999999, "/bin/sh"` where X is the address of system.
- Write down this address Y ! Now back to `showdate`, send it

```
AAAA...A + BBBB + (address X of system) + CCCC + (address Y of /bin/sh)
```

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`
- Break at `main`, then run.
- Now libraries are loaded. Find the address of `system`.
- Use `find X, +999999999, "/bin/sh"` where X is the address of system.
- Write down this address Y ! Now back to `showdate`, send it

  `AAAA...A + BBBB + (address X of system) + CCCC + (address Y of /bin/sh)`

- What about CCCC?

Return-oriented programming

1.Abusing libraries: `ret2libc`

Task:

- Load the `hello` program you made in C in `gdb`
- Break at `main`, then run.
- Now libraries are loaded. Find the address of `system`.
- Use `find X, +999999999, "/bin/sh"` where X is the address of system.
- Write down this address Y ! Now back to `showdate`, send it

  `AAAA...A + BBBB + (address X of system) + CCCC + (address Y of /bin/sh)`

- What about CCCC?

(note: might work visibly or not)

Return-oriented programming

2. Chaining

Task:

Return-oriented programming

2. Chaining

Task:

- Abuse `hijack` to make it print multiple times the same message

Return-oriented programming

2. Chaining

Task:

- Abuse `hijack` to make it print multiple times the same message
- Can we use any program or library function?

Return-oriented programming

2. Chaining

Task:

- Abuse `hijack` to make it print multiple times the same message
- Can we use any program or library function? Yes!

Return-oriented programming

2. Chaining

Task:

- Abuse `hijack` to make it print multiple times the same message
- Can we use any program or library function? Yes!
- Abuse `chaining_simple` to print messages

Return-oriented programming

2. Chaining

Task:

- Abuse `hijack` to make it print multiple times the same message
- Can we use any program or library function? Yes!
- Abuse `chaining_simple` to print messages
- Are we limited to existing functions?

Return-oriented programming

2. Chaining

Task:

- Abuse `hijack` to make it print multiple times the same message
- Can we use any program or library function? Yes!
- Abuse `chaining_simple` to print messages
- Are we limited to existing functions? No!

Return-oriented programming

3. Gadged chaining

Hard (optional) task:

Hard (optional) task:

- Read, understand and compile `chaining.c`

Hard (optional) task:

- Read, understand and compile `chaining.c`
- Find with `objdump` a sequence `pop ret` and `pop pop ret`.

Return-oriented programming

3. Gadged chaining

Hard (optional) task:

- Read, understand and compile `chaining.c`
- Find with `objdump` a sequence `pop ret` and `pop pop ret`.
- What do they do? They are called *gadgets*.

Return-oriented programming

3. Gadged chaining

Hard (optional) task:

- Read, understand and compile `chaining.c`
- Find with `objdump` a sequence `pop ret` and `pop pop ret`.
- What do they do? They are called *gadgets*.
- How can we chain the whole thing?

Return-oriented programming

3. Gadged chaining

Hard (optional) task:

- Read, understand and compile `chaining.c`
- Find with `objdump` a sequence `pop ret` and `pop pop ret`.
- What do they do? They are called *gadgets*.
- How can we chain the whole thing?

```
(exec_string)
 + 0x0badf00d + 0xcafebabe + (pop pop ret) + (add_sh)
 + 0xdeafbeef + (pop ret) + (add_bin)
 + BBBB + AAA...AA
```

Hard (optional) task:

- Read, understand and compile `chaining.c`
- Find with `objdump` a sequence `pop ret` and `pop pop ret`.
- What do they do? They are called *gadgets*.
- How can we chain the whole thing?

```
(exec_string)
 + 0x0badf00d + 0xcafebabe + (pop pop ret) + (add_sh)
 + 0xdeafbeef + (pop ret) + (add_bin)
 + BBBB + AAA...AA
```

- Make a python program `chaining.py` doing that

Return-oriented programming

3. Gadged chaining

Hard (optional) task:

- Read, understand and compile `chaining.c`
- Find with `objdump` a sequence `pop ret` and `pop pop ret`.
- What do they do? They are called *gadgets*.
- How can we chain the whole thing?

```
(exec_string)
 + 0x0badf00d + 0xcafebabe + (pop pop ret) + (add_sh)
 + 0xdeafbeef + (pop ret) + (add_bin)
 + BBBB + AAA...AA
```

- Make a python program `chaining.py` doing that
- Hint: use `struct.pack("I", address)`

Return-oriented programming

3. Gadged chaining

Hard (optional) task:

- Read, understand and compile `chaining.c`
- Find with `objdump` a sequence `pop ret` and `pop pop ret`.
- What do they do? They are called *gadgets*.
- How can we chain the whole thing?

```
(exec_string)
 + 0x0badf00d + 0xcafebabe + (pop pop ret) + (add_sh)
 + 0xdeafbeef + (pop ret) + (add_bin)
 + BBBB + AAA...AA
```

- Make a python program `chaining.py` doing that
- Hint: use `struct.pack("I", address)`

Return-oriented programming

4. The remaining details

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

  Everything (Turing-complete, Dolan 2013)

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

  Everything (Turing-complete, Dolan 2013)

- What about ASLR?

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

  Everything (Turing-complete, Dolan 2013)

- What about ASLR?

  If `libc` isn't randomised, ASLR does nothing to protect. ASLR can be bypassed by stack reading too.

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

  Everything (Turing-complete, Dolan 2013)

- What about ASLR?

  If `libc` isn't randomised, ASLR does nothing to protect. ASLR can be bypassed by stack reading too.

- Do I need the source code?

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

  Everything (Turing-complete, Dolan 2013)

- What about ASLR?

  If `libc` isn't randomised, ASLR does nothing to protect. ASLR can be bypassed by stack reading too.

- Do I need the source code?

  Not necessarily, blind ROP exists, e.g. `braille`

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

  Everything (Turing-complete, Dolan 2013)

- What about ASLR?

  If `libc` isn't randomised, ASLR does nothing to protect. ASLR can be bypassed by stack reading too.

- Do I need the source code?

  Not necessarily, blind ROP exists, e.g. `braille`

- Is it not amazing?

Return-oriented programming

4. The remaining details

- SUID `sh` doesn't work?

  That's because it's aliased to `sh -c`. That's bypassable.

- How to find gadgets?

  Automated tools e.g. `ROPgadget`

- What can we do with gadgets?

  Everything (Turing-complete, Dolan 2013)

- What about ASLR?

  If `libc` isn't randomised, ASLR does nothing to protect. ASLR can be bypassed by stack reading too.

- Do I need the source code?

  Not necessarily, blind ROP exists, e.g. `braille`

- Is it not amazing? Yes, yes it is.

Use-after-free

We saw that the "*function*" concept was troublesome.

Use-after-free

We saw that the "*function*" concept was troublesome.

In the 1970s, people had an ever crazier idea...

Use-after-free

We saw that the "*function*" concept was troublesome.

In the 1970s, people had an ever crazier idea...

..."Object-oriented" programming languages

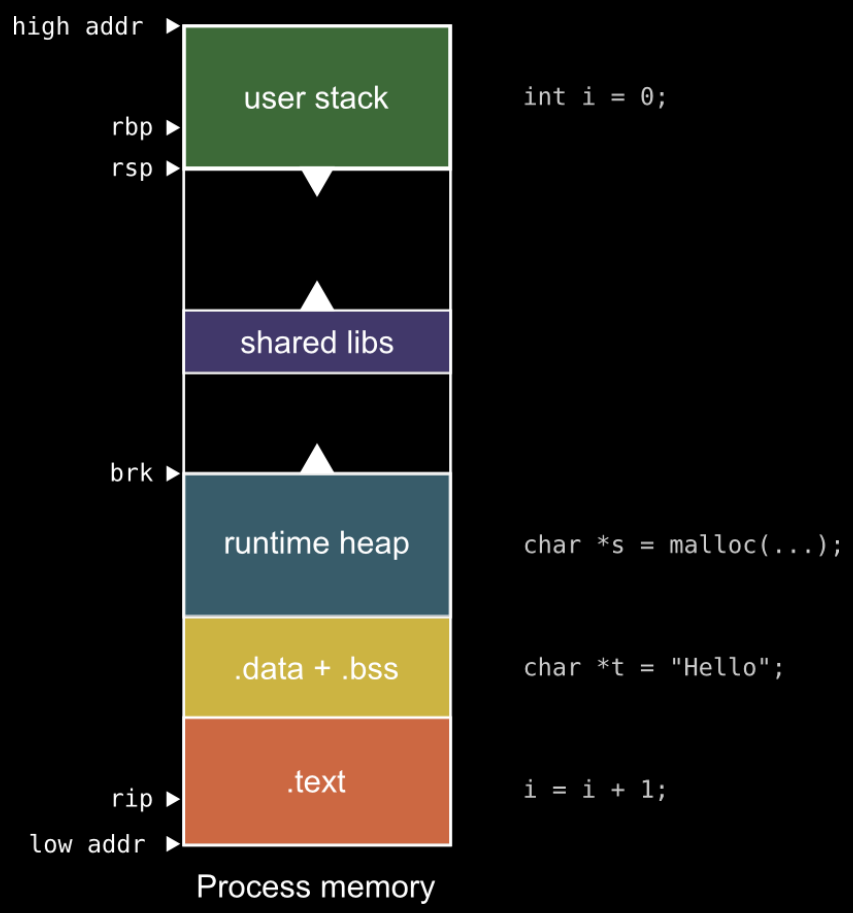
Use-after-free

We saw that the "*function*" concept was troublesome.

In the 1970s, people had an ever crazier idea...

..."Object-oriented" programming languages – OOPs ;)

Reminder: the heap



Reminder: the heap

The heap enables run-time allocation and reallocation of memory.

Reminder: the heap

The heap enables run-time allocation and reallocation of memory.

- `x = malloc(sizeof(int));`

Reminder: the heap

The heap enables run-time allocation and reallocation of memory.

- `x = malloc(sizeof(int));`
- `y = realloc(x, sizeof(int));`

Reminder: the heap

The heap enables run-time allocation and reallocation of memory.

- `x = malloc(sizeof(int));`
- `y = realloc(x, sizeof(int));`
- `free(x);`

Reminder: the heap

The heap enables run-time allocation and reallocation of memory.

- `x = malloc(sizeof(int));`
- `y = realloc(x, sizeof(int));`
- `free(x);`

There is no intrinsic order in the heap.

The stack and the heap

The stack and the heap

Heap problems

What happens when...

Heap problems

What happens when...

- What happens when we forget to `free`?

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again?

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again? Double-free

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again? Double-free
- What happens when we fail to `malloc` or `realloc`?

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again? Double-free
- What happens when we fail to `malloc` or `realloc`? Segfault

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again? Double-free
- What happens when we fail to `malloc` or `realloc`? Segfault
- What happens when we write more than we should?

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again? Double-free
- What happens when we fail to `malloc` or `realloc`? Segfault
- What happens when we write more than we should? Overflow

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again? Double-free
- What happens when we fail to `malloc` or `realloc`? Segfault
- What happens when we write more than we should? Overflow
- What happens when we use what we `freeed`?

Heap problems

What happens when...

- What happens when we forget to `free`? Memory leaks
- What happens when we `free` again? Double-free
- What happens when we fail to `malloc` or `realloc`? Segfault
- What happens when we write more than we should? Overflow
- What happens when we use what we `freeed`? UAF.

Use-after-free

What happens is that we try to use a value that was previously freed.

Use-after-free

What happens is that we try to use a value that was previously freed.

- It might be data controlled by another program, e.g. an attacker

Use-after-free

What happens is that we try to use a value that was previously freed.

- It might be data controlled by another program, e.g. an attacker
- Oh, and remember, data is code.

Use-after-free

What happens is that we try to use a value that was previously freed.

- It might be data controlled by another program, e.g. an attacker
- Oh, and remember, data is code.

Object-oriented constructions are especially vulnerable.

The `vtable`

- A C++ class has an array, called a *virtual table*, which contains pointers to all the members of the class (attributes and methods).

The vtable

- A C++ class has an array, called a *virtual table*, which contains pointers to all the members of the class (attributes and methods).
- This vtable is reconstructed at run-time (inheritance, polymorphism, object creation)

The `vtable`

- A C++ class has an array, called a *virtual table*, which contains pointers to all the members of the class (attributes and methods).
- This vtable is reconstructed at run-time (inheritance, polymorphism, object creation)
- Calling a method means running what the vtable says

The `vtable`

- A C++ class has an array, called a *virtual table*, which contains pointers to all the members of the class (attributes and methods).
- This vtable is reconstructed at run-time (inheritance, polymorphism, object creation)
- Calling a method means running what the vtable says

Control the `vtable`, control the program.

From UAF to ACE

The strategy for exploiting an UAF vulnerability is simple:

From UAF to ACE

The strategy for exploiting an UAF vulnerability is simple:

- *Spray the heap*: allocate as much as possible, hoping it'll replace the freed object

From UAF to ACE

The strategy for exploiting an UAF vulnerability is simple:

- *Spray the heap*: allocate as much as possible, hoping it'll replace the freed object
- *Run* the vulnerable program, hoping it'll trigger the UAF object

From UAF to ACE

The strategy for exploiting an UAF vulnerability is simple:

- *Spray the heap*: allocate as much as possible, hoping it'll replace the freed object
- *Run* the vulnerable program, hoping it'll trigger the UAF object
- *Exploit*, you are now in control of eip. Use either the shellcode way, or the ROP way.

From UAF to ACE

The strategy for exploiting an UAF vulnerability is simple:

- *Spray the heap*: allocate as much as possible, hoping it'll replace the freed object
- *Run* the vulnerable program, hoping it'll trigger the UAF object
- *Exploit*, you are now in control of eip. Use either the shellcode way, or the ROP way.

Especially vulnerable are modern Internet browsers.

From UAF to ACE

The strategy for exploiting an UAF vulnerability is simple:

- *Spray the heap*: allocate as much as possible, hoping it'll replace the freed object
- *Run* the vulnerable program, hoping it'll trigger the UAF object
- *Exploit*, you are now in control of eip. Use either the shellcode way, or the ROP way.

Especially vulnerable are modern Internet browsers.

Because they feed on memory like nerds on lingerie.

From UAF to ACE

A real-world example: IE 6 to 11 (CVE-2014-1776)

From UAF to ACE

A real-world example: IE 6 to 11 (CVE-2014-1776)

- Object used in `MSHTML!CMarkup::IsConnectedToPrimaryMarkup`

From UAF to ACE

A real-world example: IE 6 to 11 (CVE-2014-1776)

- Object used in `MSHTML!CMarkup::IsConnectedToPrimaryMarkup`
- But was freed before by `MSHTML!Cmarkup::PrivateRelease`

From UAF to ACE

A real-world example: IE 6 to 11 (CVE-2014-1776)

- Object used in `MSHTML!CMarkup::IsConnectedToPrimaryMarkup`
- But was freed before by `MSHTML!Cmarkup::PrivateRelease`
- Object size is `0x428`

From UAF to ACE

A real-world example: IE 6 to 11 (CVE-2014-1776)

- Object used in `MSHTML!CMarkup::IsConnectedToPrimaryMarkup`
- But was freed before by `MSHTML!Cmarkup::PrivateRelease`
- Object size is `0x428`
- In the wild, exploits used Flash to spray the heap.

From UAF to ACE

A real-world example: IE 6 to 11 (CVE-2014-1776)

- Object used in `MSHTML!CMarkup::IsConnectedToPrimaryMarkup`
- But was freed before by `MSHTML!Cmarkup::PrivateRelease`
- Object size is `0x428`
- In the wild, exploits used Flash to spray the heap.
- But can be done in pure HTML+JavaScript

Where do we find UAF?

Take an example (Adobe Flash). Just this year,

Where do we find UAF?

Take an example (Adobe Flash). Just this year,

CVE-2015-0308, CVE-2015-0311, CVE-2015-0312,
CVE-2015-0313, CVE-2015-0315, CVE-2015-0320,
CVE-2015-0322, ..., CVE-2015-5119, CVE-2015-5122

All exploited in the wild.

Exercise: make your own. Jung Hoon Lee (17yrs) earned $225,000 at Pwn2Own 2015.