Nhacker: A Neural Hacker

Enrique Sanchez
enrique.sanchez@yaguarete-sec.com

X'Con2006

First things First

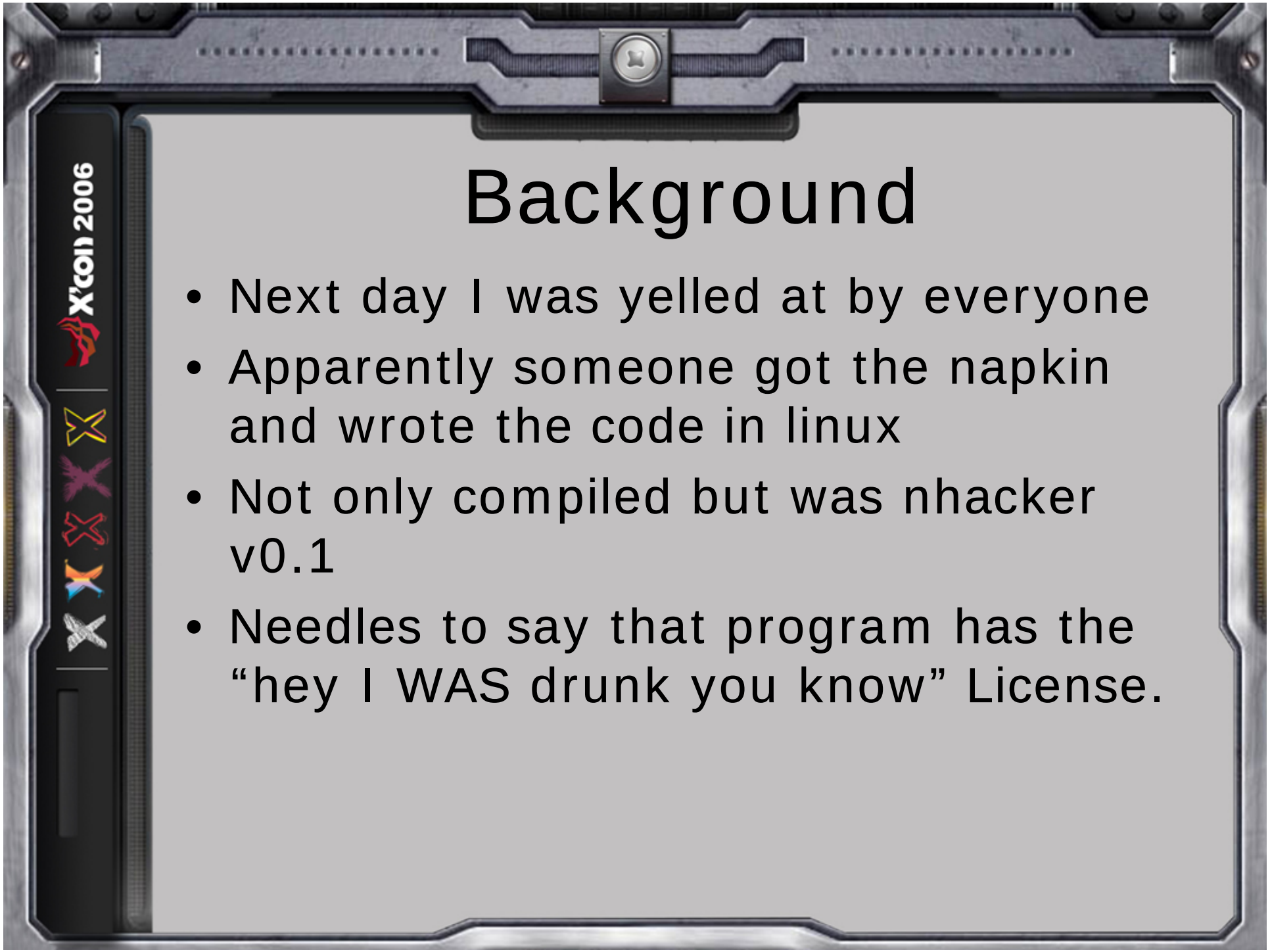
- I'm not an expert in neural networks.
- I've been known to be mistaken, quite often.
- If you have 999,999 monkeys write in a typewriter you can probably write something as ugly as DCOM in around 5 years.
- If I can read C, hell why my programs
- couldn't!!!

Neural Networks

- Most of the times they are mistaken by Expert Systems
- Nice since they tend to learn.
- Fuzzy logic can be introduced.
- At the end you only have to show them what is good or bad, the in between is not important for you (as the guy who coded it)

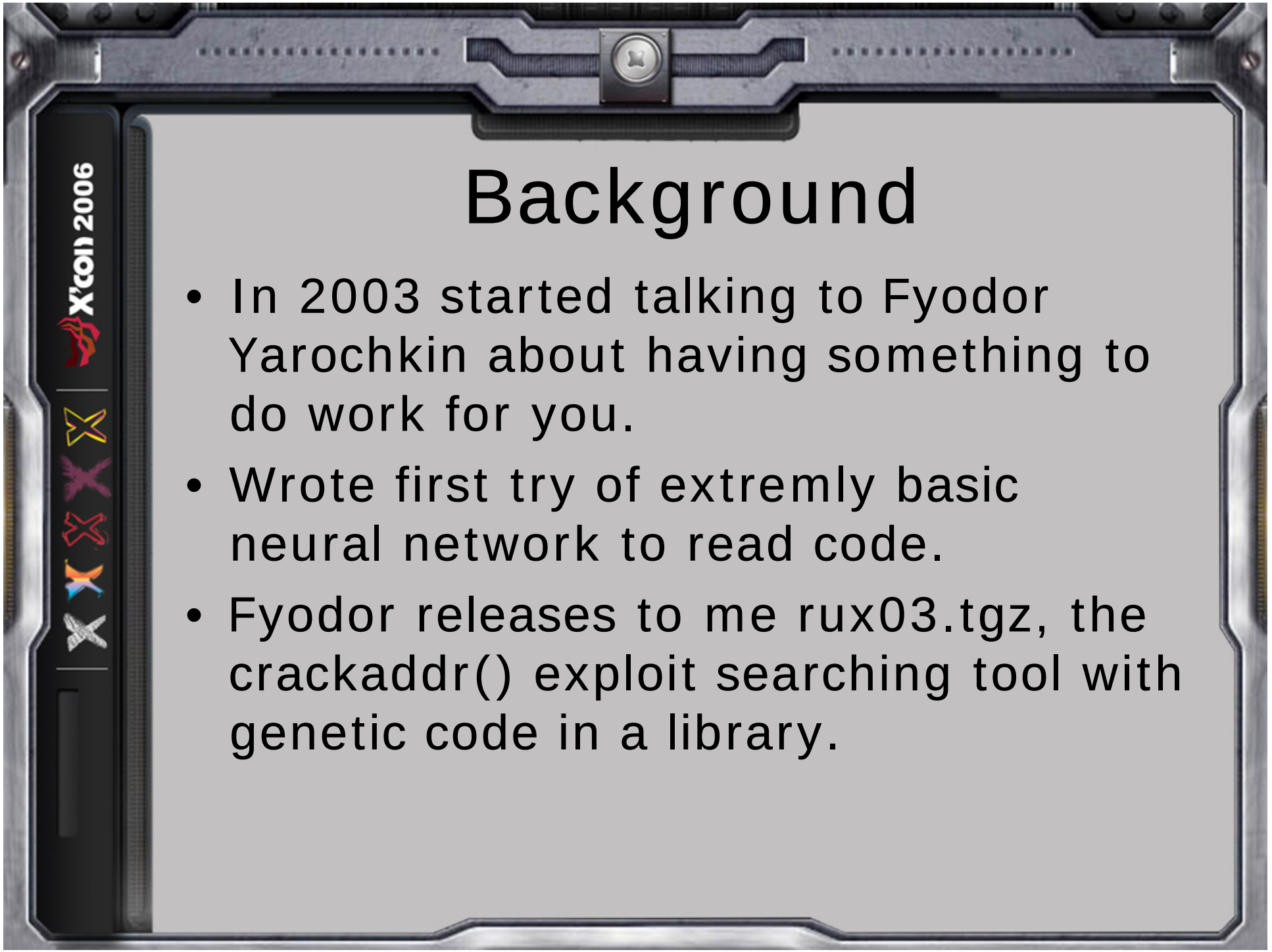
Background

- Everything started while getting drunk (Don't all good ideas start like that?)
- I started talking to friends on how it would be cool to code yourself into a protocol or the internet.
  - C After a few laughs and “you can't write that!”
  - C Remember writting some gibberish on a napkin



Background

- Next day I was yelled at by everyone
- Apparently someone got the napkin and wrote the code in linux
- Not only compiled but was nhacker v0.1
- Needles to say that program has the “hey I WAS drunk you know” License.

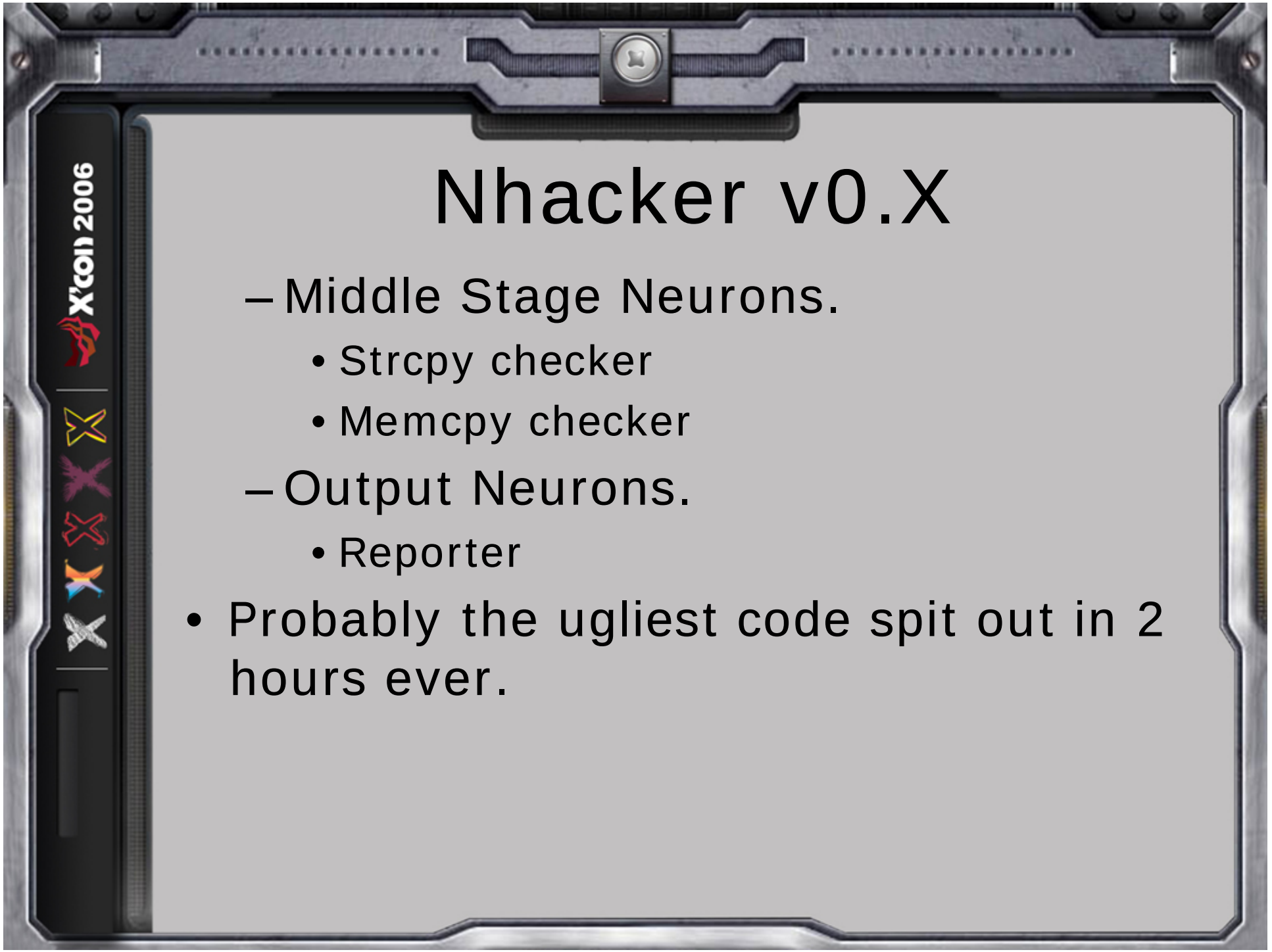


Background

- In 2003 started talking to Fyodor Yarochkin about having something to do work for you.
- Wrote first try of extremly basic neural network to read code.
- Fyodor releases to me rux03.tgz, the crackaddr() exploit searching tool with genetic code in a library.

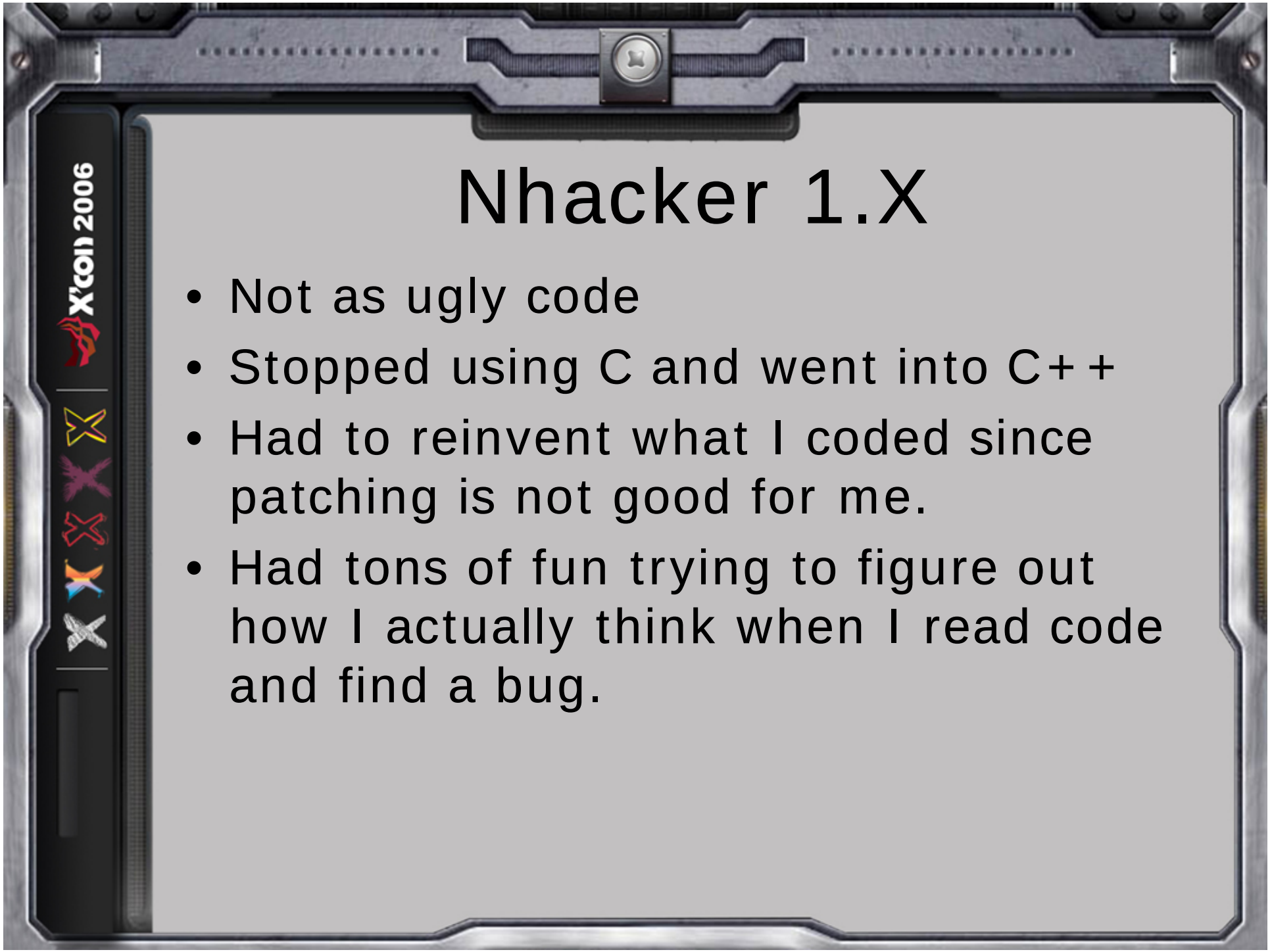
Nhacker v0.X

- Version never left my computer.
- The outline of this very very basic:
  - Input Neurons.
    - File reader
    - Line reader
    - Function parser
    - Variable composer
  - Middle Stage Neurons.
  - Output Neurons.



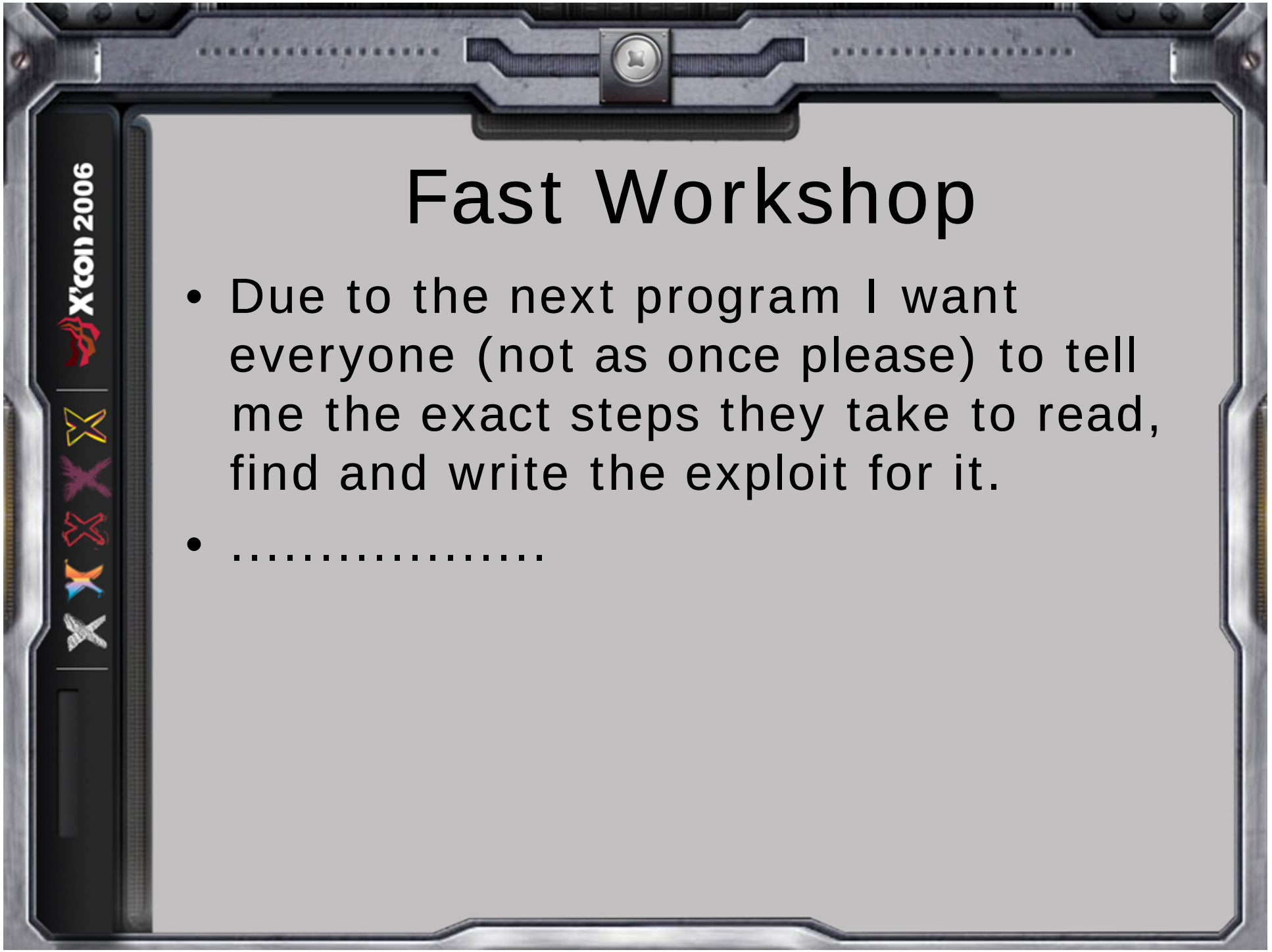
Nhacker v0.X

- Middle Stage Neurons.
  - Strcpy checker
  - Memcpy checker
- Output Neurons.
  - Reporter
- Probably the ugliest code spit out in 2 hours ever.



Nhacker 1.X

- Not as ugly code
- Stopped using C and went into C++
- Had to reinvent what I coded since patching is not good for me.
- Had tons of fun trying to figure out how I actually think when I read code and find a bug.



Fast Workshop

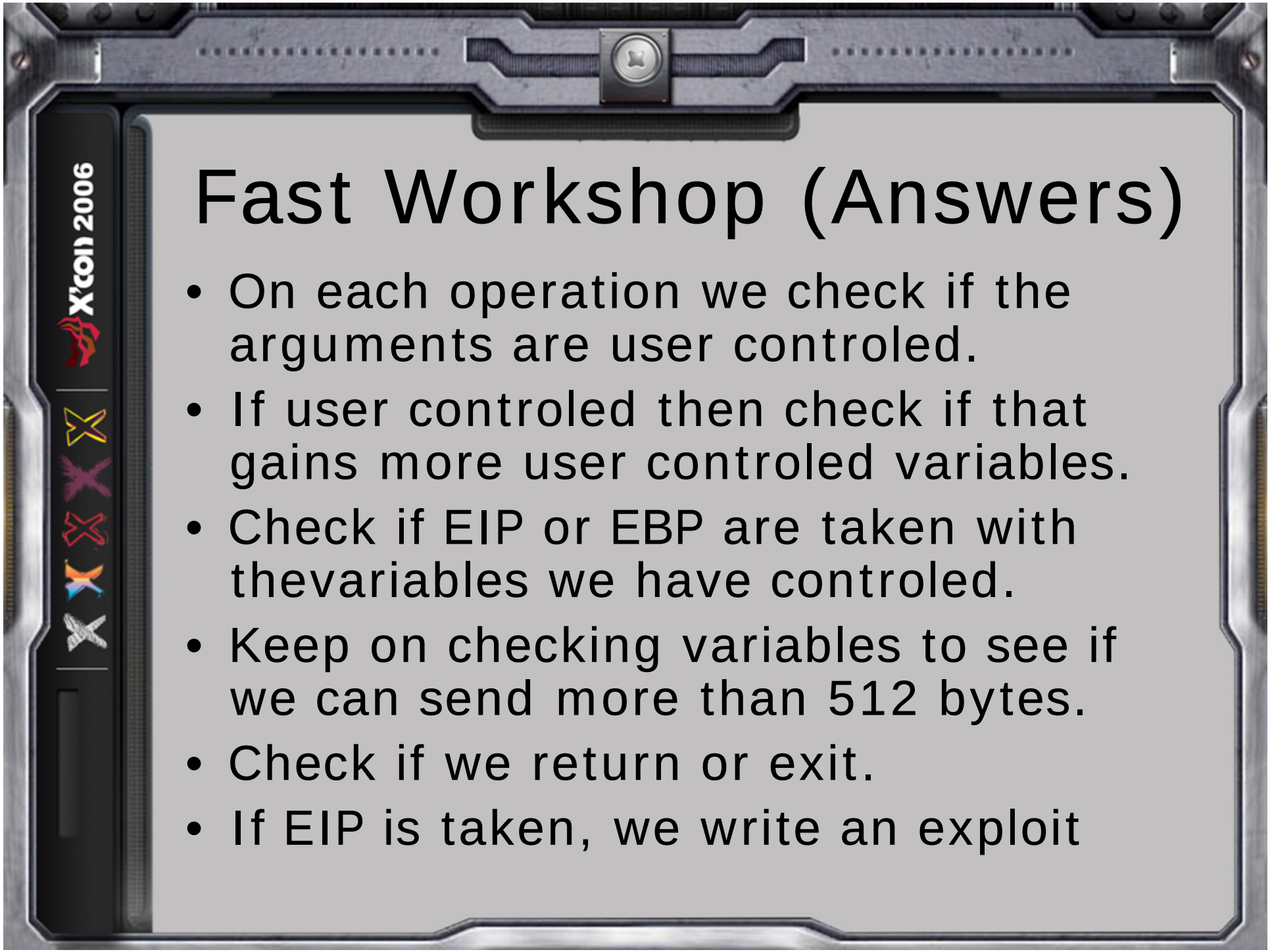
- Due to the next program I want everyone (not as once please) to tell me the exact steps they take to read, find and write the exploit for it.
- .....

Fast Workshop (Example)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    char buff[512];
    if(argc < 2)
    exit;
    strcpy(buff, argv[1]);
    printf("Man this is VERY basic: %s\n", buff);
    return 0;
}
```

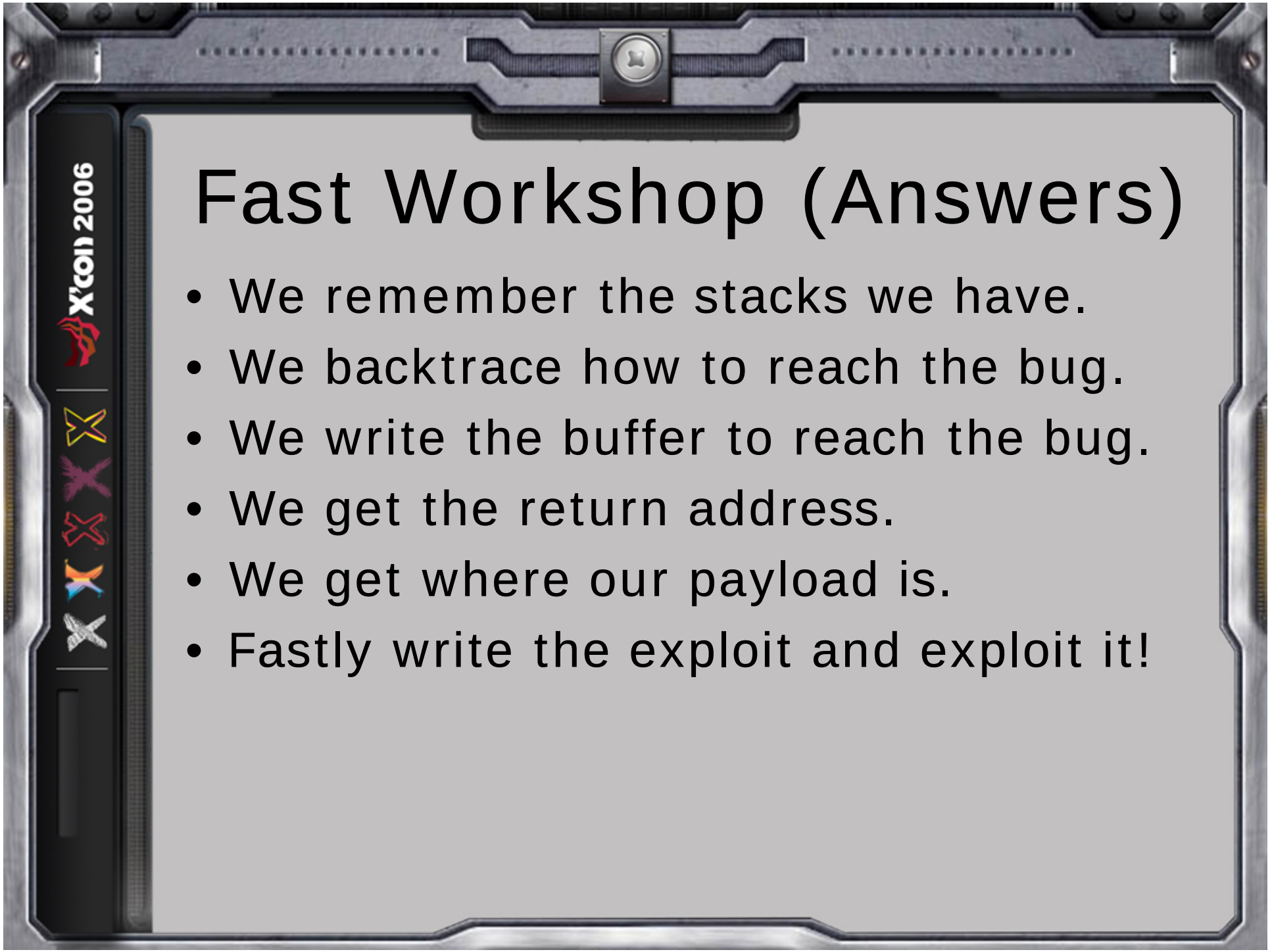
Fast Workshop (Answers)

- This are the (almost) complete steps:
  - Find the file.
  - Check if you have more files in the directory.
  - Open the file.
  - Read each line looking for functions.
  - On each function create a stack for it.
  - Fill out the stack and retain it in memory to “enter” it when is called.
  - Populate variables with sizes.



Fast Workshop (Answers)

- On each operation we check if the arguments are user controled.
- If user controled then check if that gains more user controled variables.
- Check if EIP or EBP are taken with thevariables we have controled.
- Keep on checking variables to see if we can send more than 512 bytes.
- Check if we return or exit.
- If EIP is taken, we write an exploit

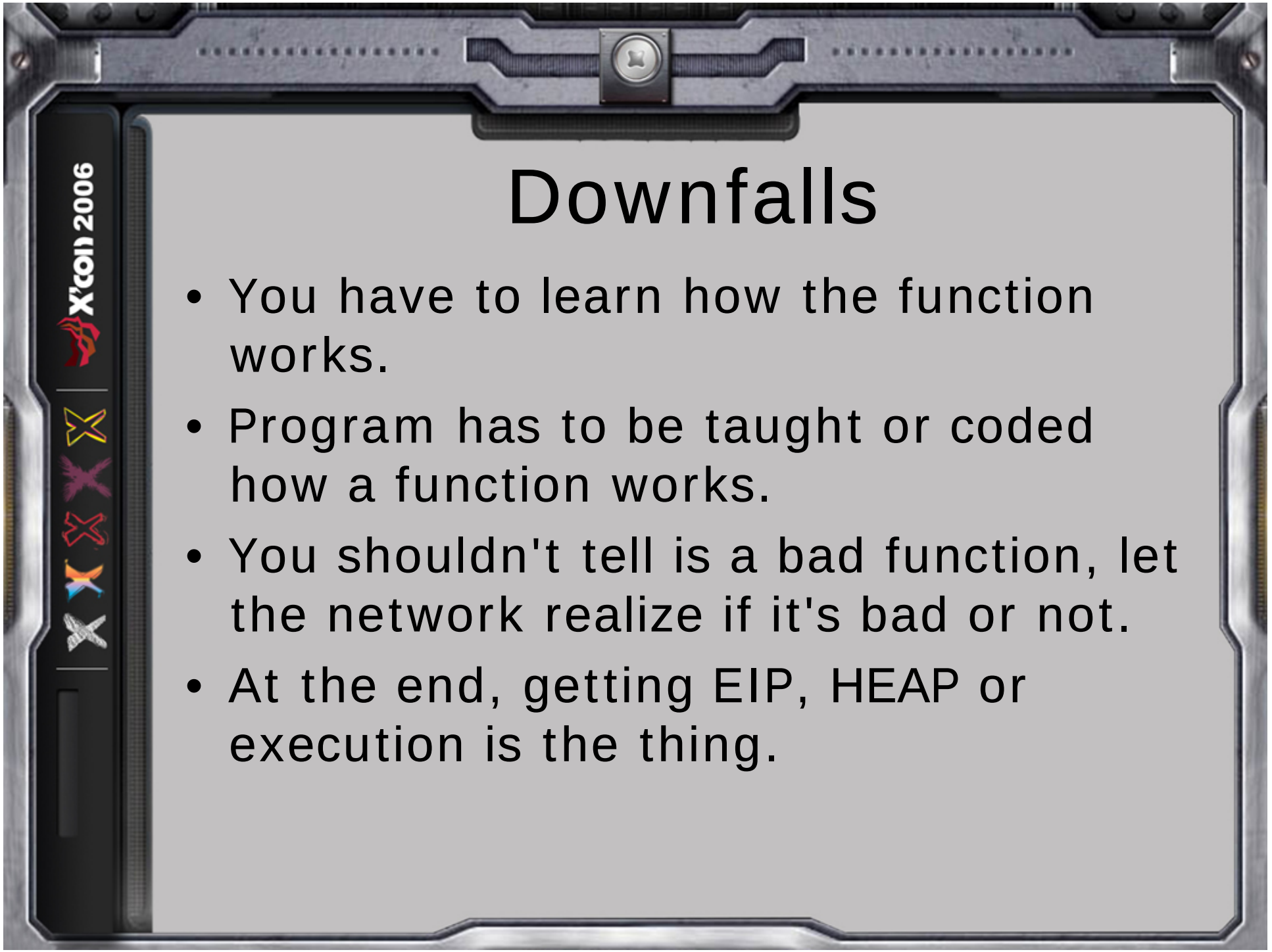


Fast Workshop (Answers)

- We remember the stacks we have.
- We backtrace how to reach the bug.
- We write the buffer to reach the bug.
- We get the return address.
- We get where our payload is.
- Fastly write the exploit and exploit it!

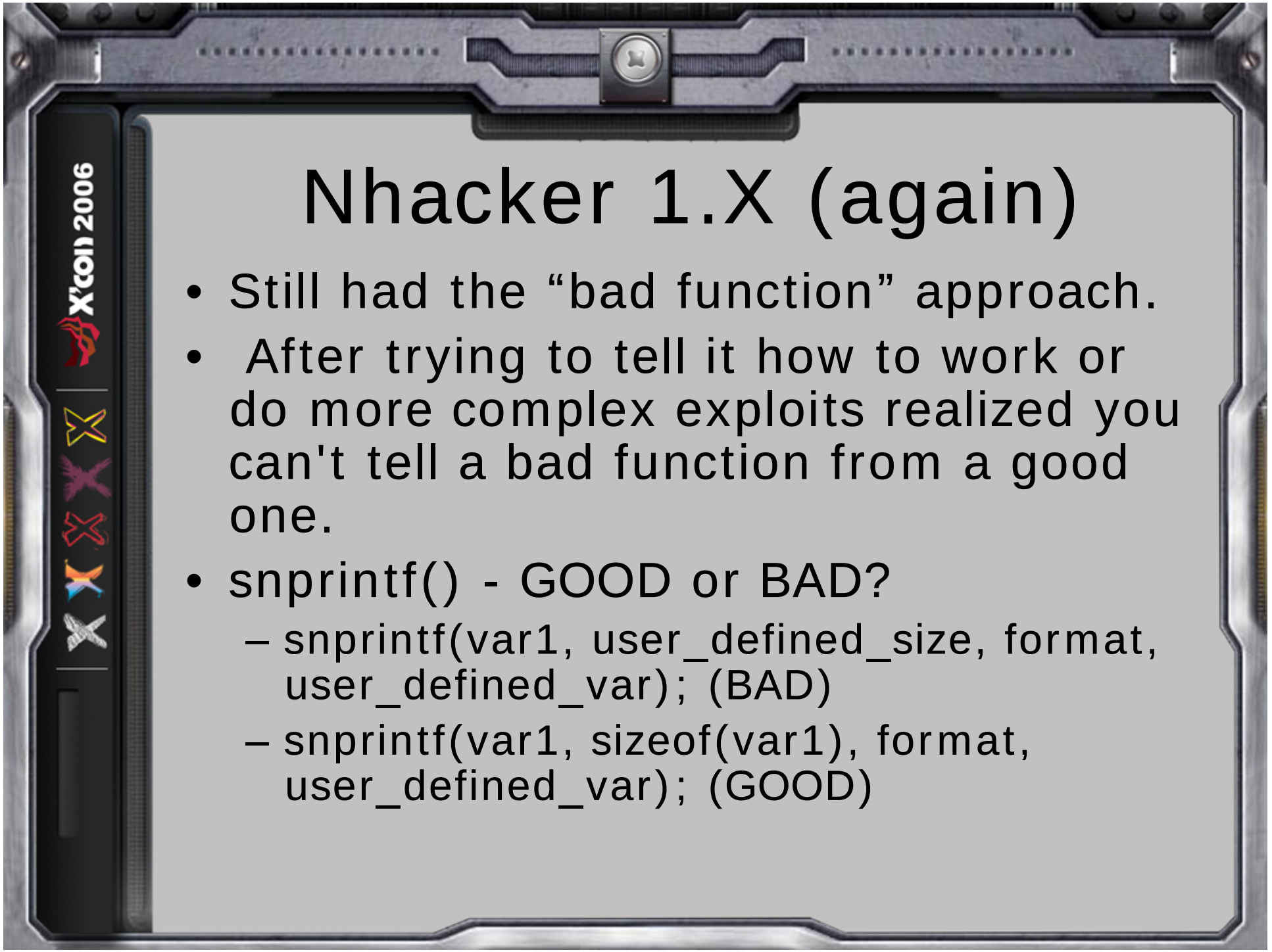
Fast Workshop (Answers)

- Only 19 steps!
- Imagine how is to do samba, or openssh, or apache.
- After 2 hours brain shuts down.
- After 2 hours computer still works.
- BAD: The program is as smart as the
- coder? (NOT REALLY THANK GOD)



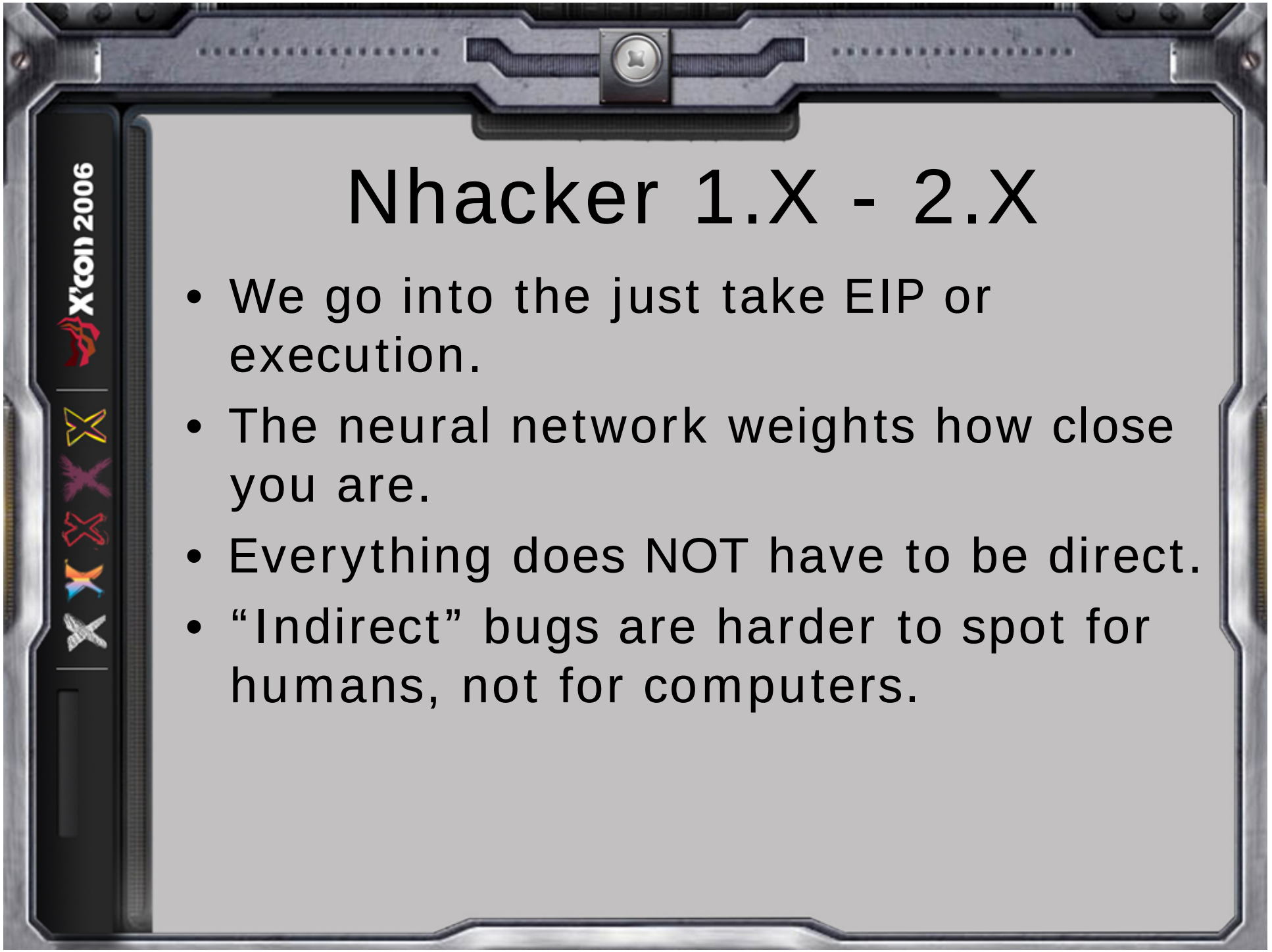
Downfalls

- You have to learn how the function works.
- Program has to be taught or coded how a function works.
- You shouldn't tell is a bad function, let the network realize if it's bad or not.
- At the end, getting EIP, HEAP or execution is the thing.



Nhacker 1.X (again)

- Still had the “bad function” approach.
- After trying to tell it how to work or do more complex exploits realized you can't tell a bad function from a good one.
- snprintf() - GOOD or BAD?
  - snprintf(var1, user_defined_size, format, user_defined_var); (BAD)
  - snprintf(var1, sizeof(var1), format, user_defined_var); (GOOD)

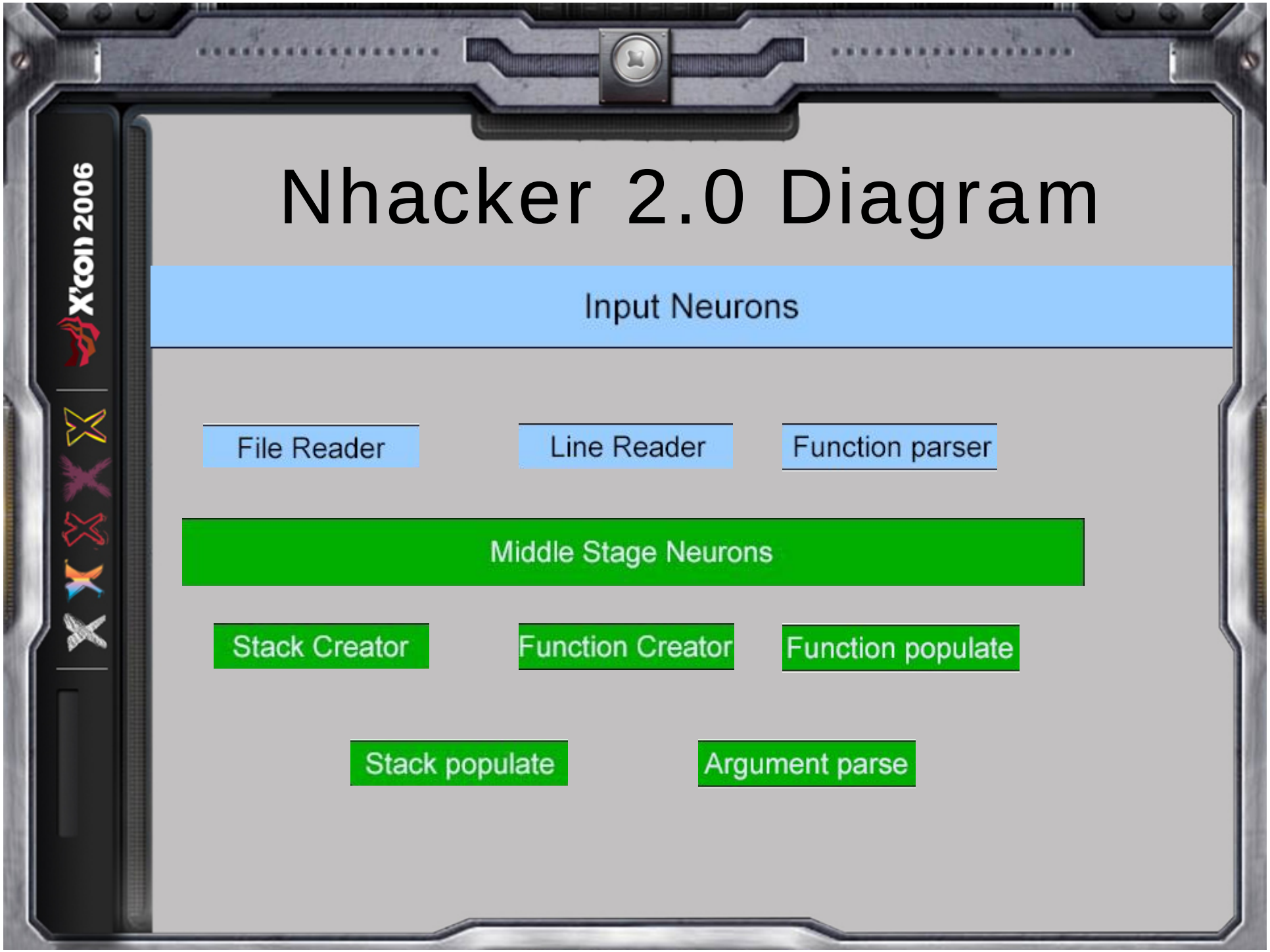


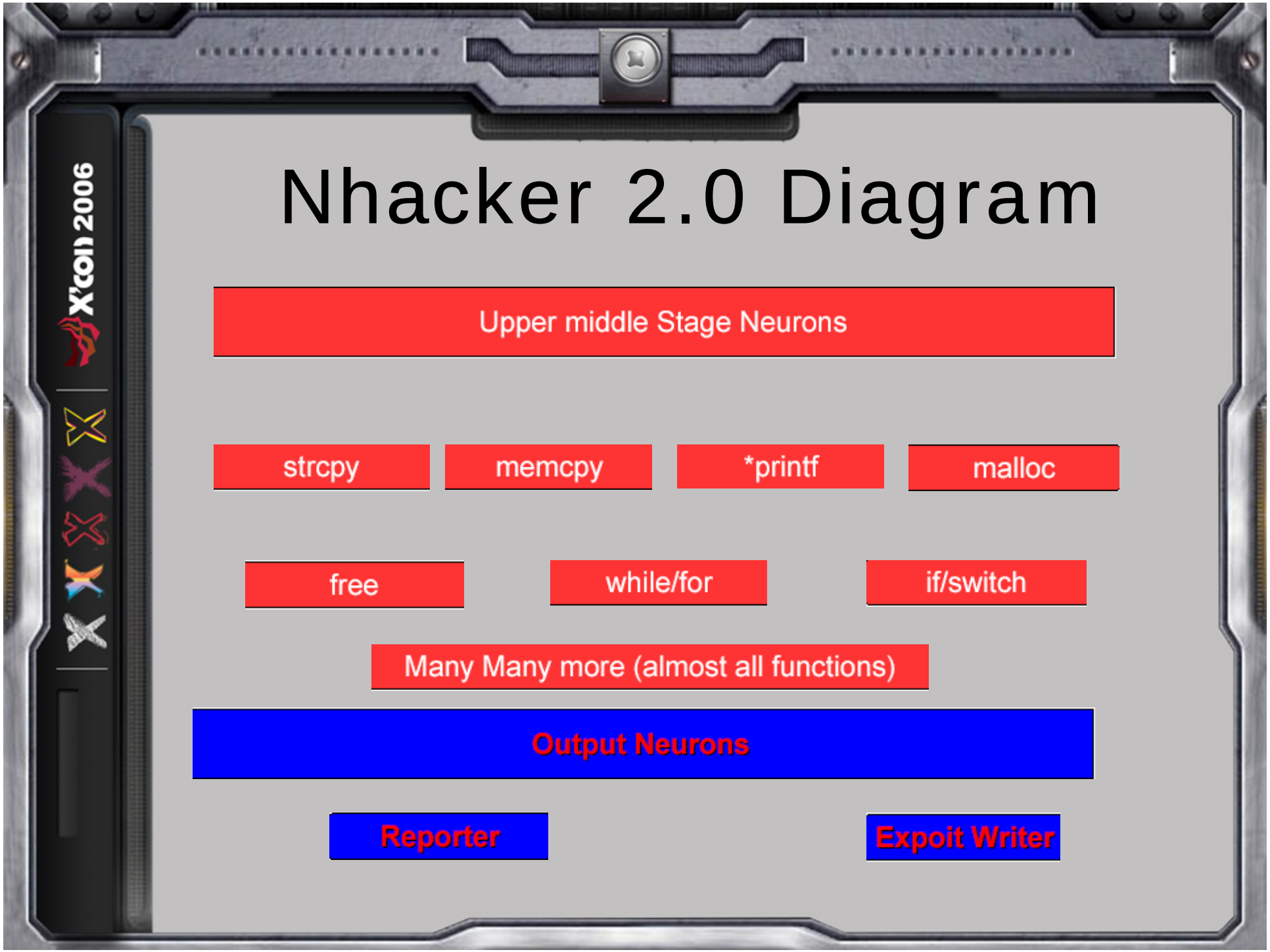
Nhacker 1.X - 2.X

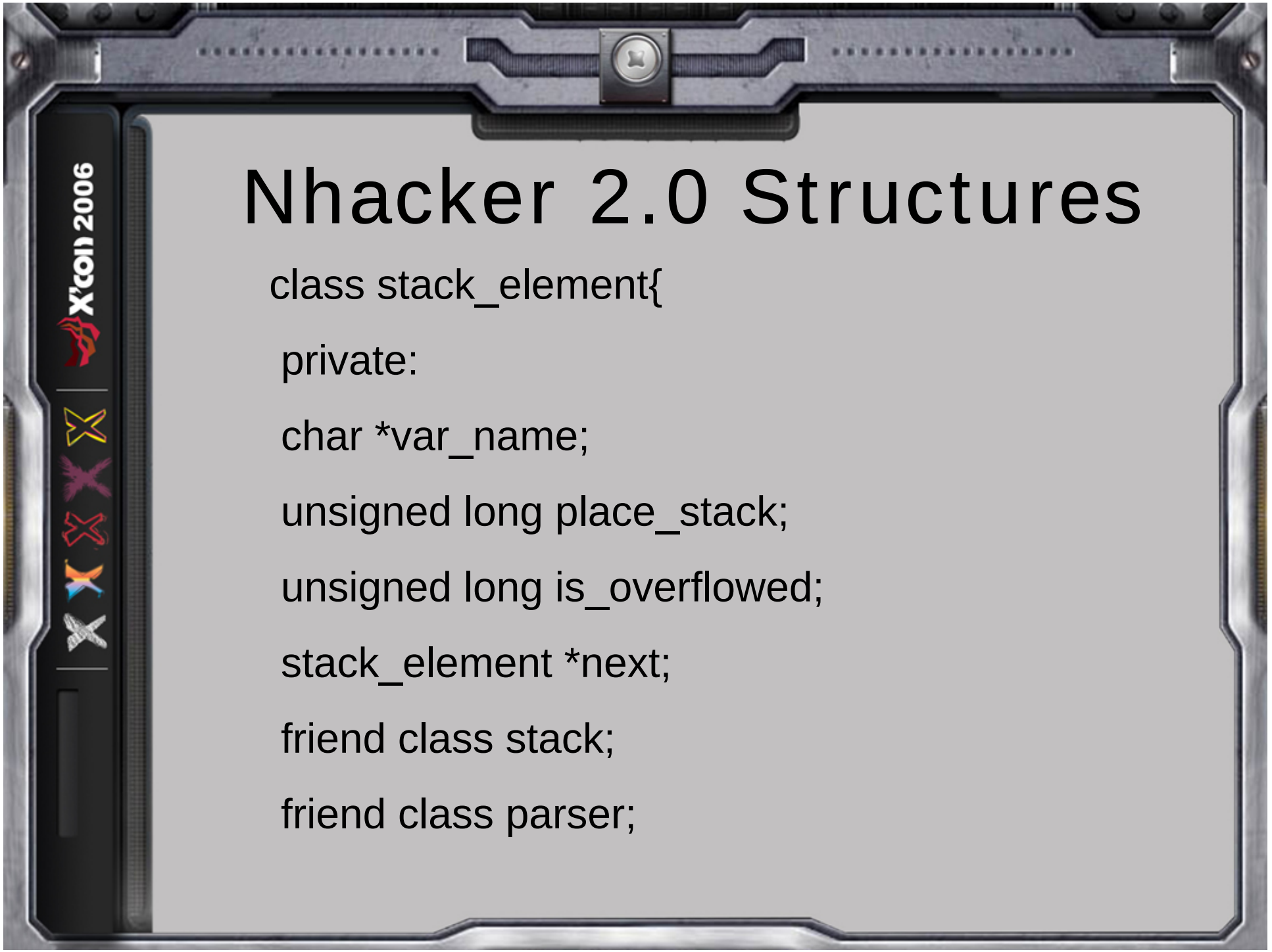
- We go into the just take EIP or execution.
- The neural network weights how close you are.
- Everything does NOT have to be direct.
- “Indirect” bugs are harder to spot for humans, not for computers.

Nhacker 2.X Characteristics

- It has more neurons.
- More complex.
- Way more efficient.
- Not faster than version 1.X but more on real situations.
- It has 7 0hday on the list.

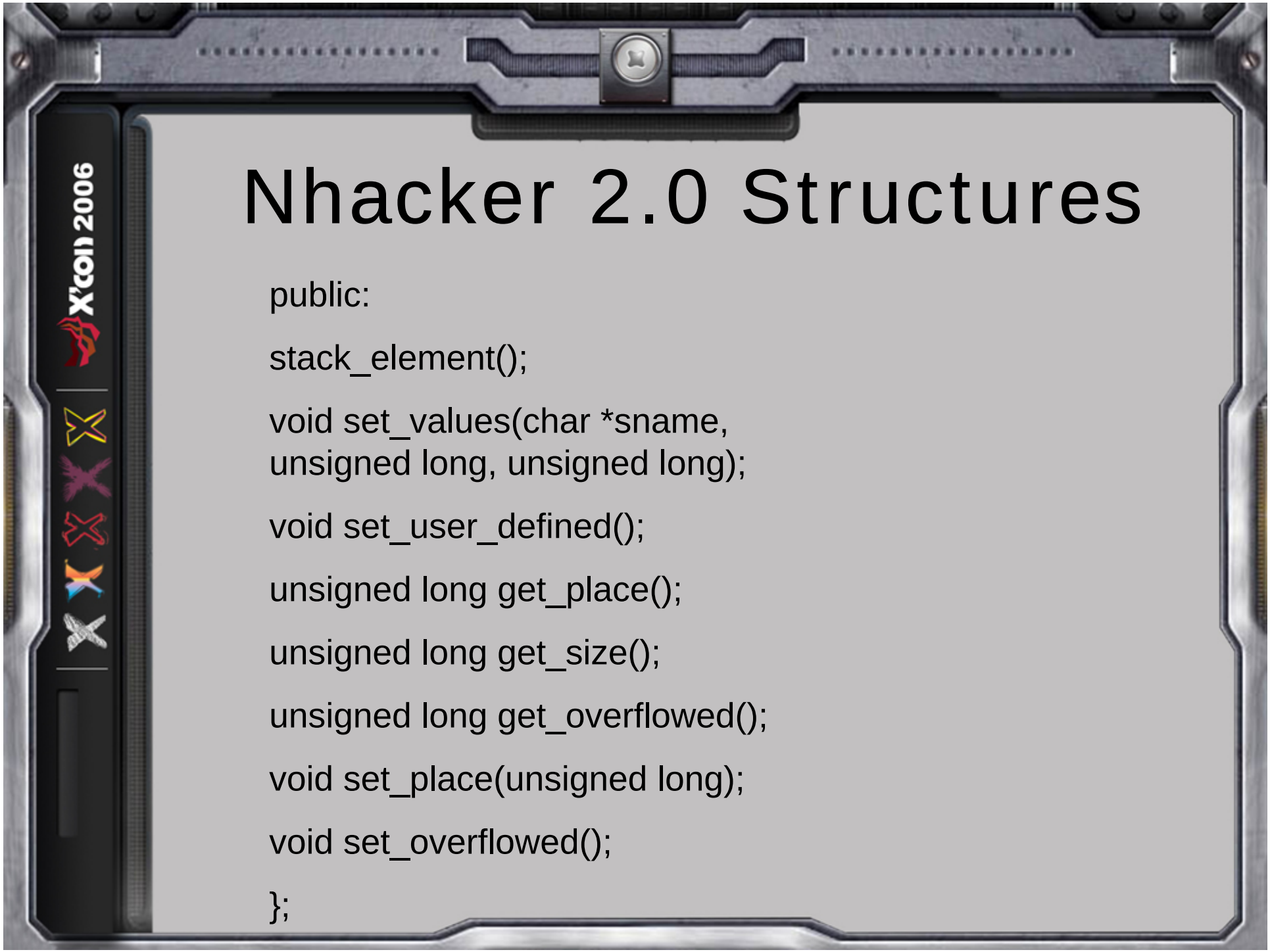

X'con 2006
Nhacker 2.0 Diagram
Input Neurons
File Reader
Line Reader
Function parser
Middle Stage Neurons
Stack Creator
Function Creator
Function populate
Stack populate
Argument parse


X'con 2006
Nhacker 2.0 Diagram
Upper middle Stage Neurons
strcpy
memcpy
*printf
malloc
free
while/for
if/switch
Many Many more (almost all functions)
Output Neurons
Reporter
Expoit Writer



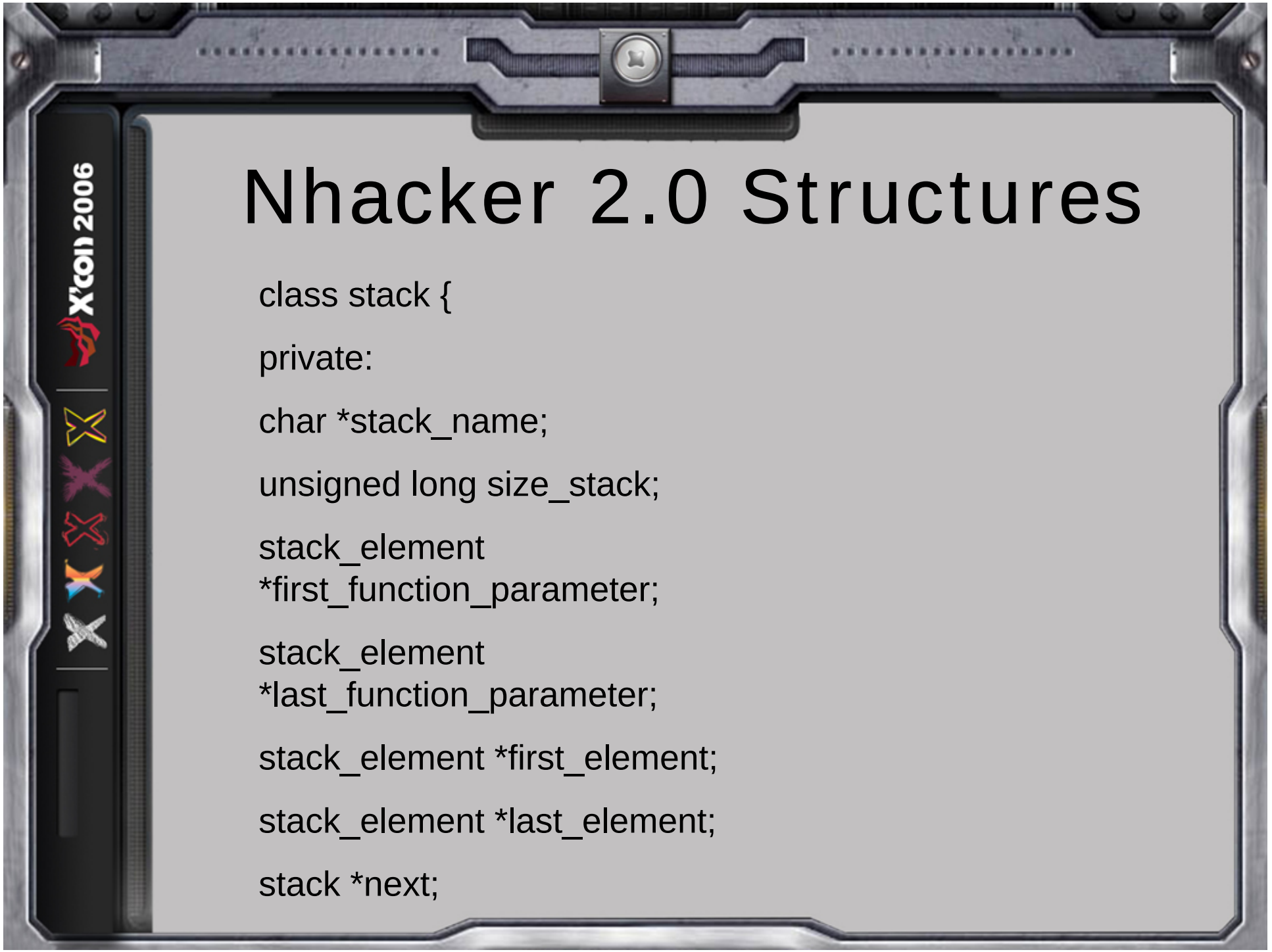
Nhacker 2.0 Structures

```
class stack_element{
 private:
 char *var_name;
 unsigned long place_stack;
 unsigned long is_overflowed;
 stack_element *next;
 friend class stack;
 friend class parser;
```



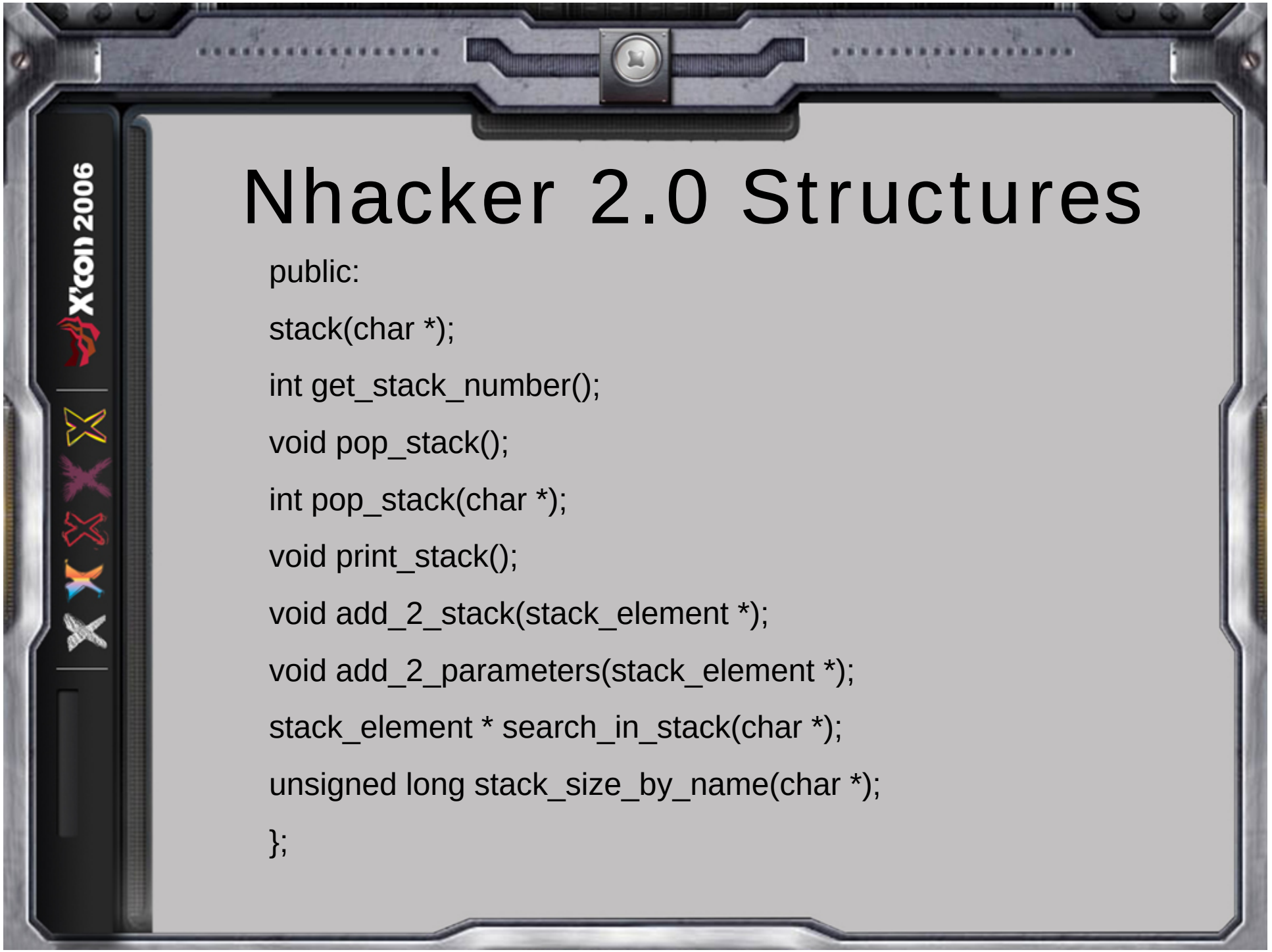
Nhacker 2.0 Structures

```
public:

stack_element();

void set_values(char *sname,
unsigned long, unsigned long);

void set_user_defined();

unsigned long get_place();

unsigned long get_size();

unsigned long get_overflowed();

void set_place(unsigned long);

void set_overflowed();

};
```



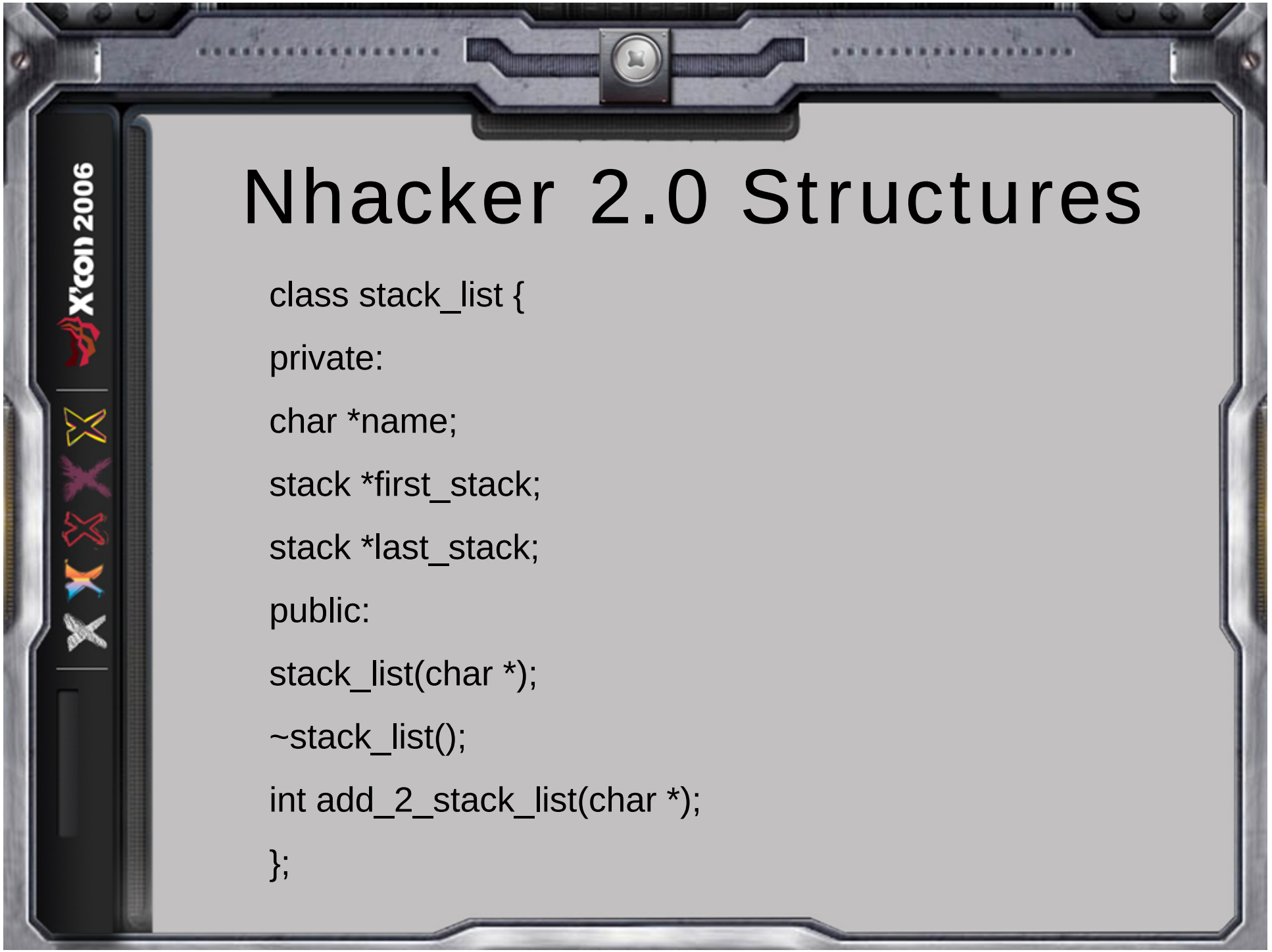
Nhacker 2.0 Structures

```
class stack {

private:

char *stack_name;

unsigned long size_stack;

stack_element
*first_function_parameter;

stack_element
*last_function_parameter;

stack_element *first_element;

stack_element *last_element;

stack *next;
```



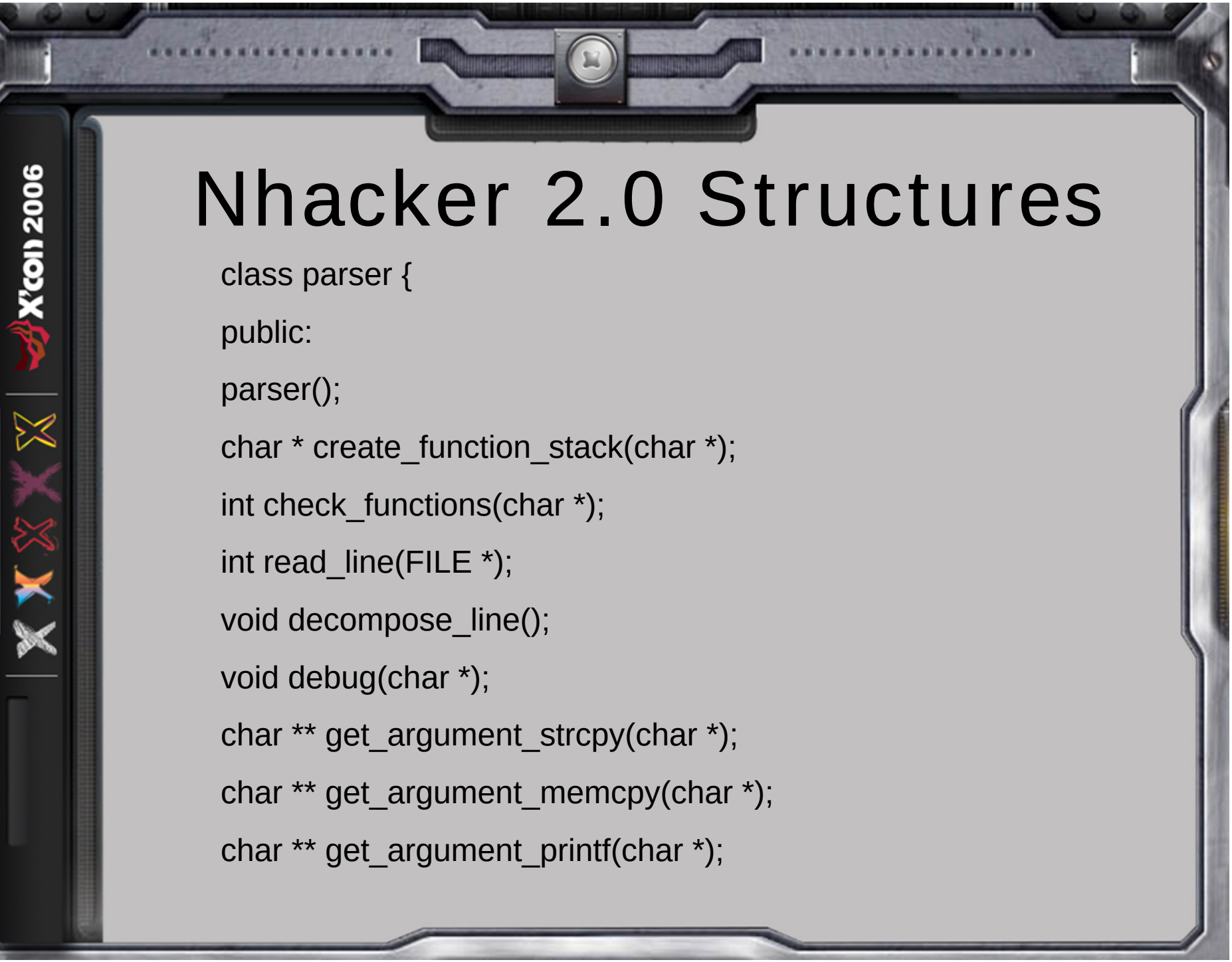
Nhacker 2.0 Structures

```
public:
stack(char *);
int get_stack_number();
void pop_stack();
int pop_stack(char *);
void print_stack();
void add_2_stack(stack_element *);
void add_2_parameters(stack_element *);
stack_element * search_in_stack(char *);
unsigned long stack_size_by_name(char *);
};
```



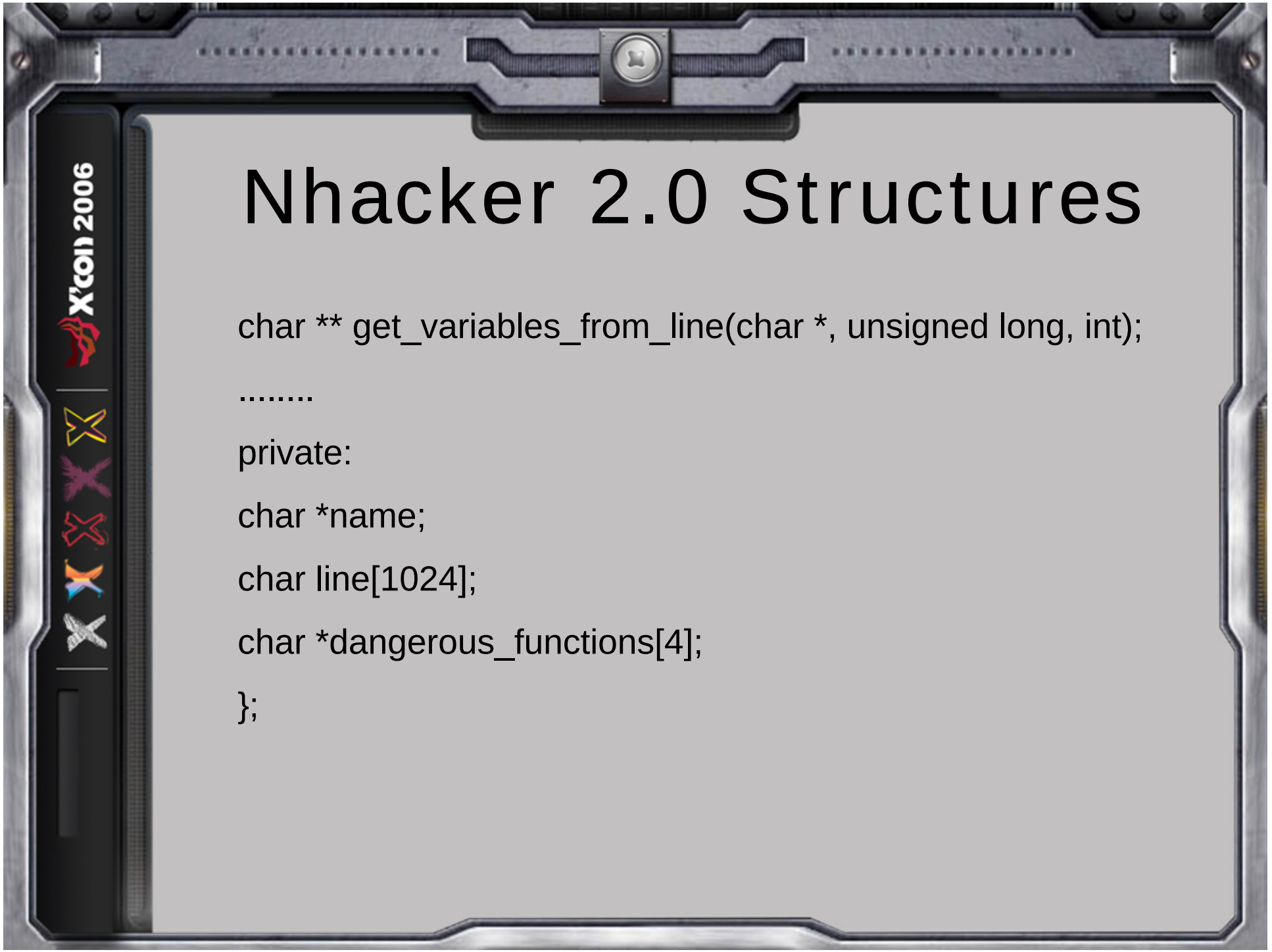
Nhacker 2.0 Structures

```
class stack_list {
private:
char *name;
stack *first_stack;
stack *last_stack;
public:
stack_list(char *);
~stack_list();
int add_2_stack_list(char *);
};
```



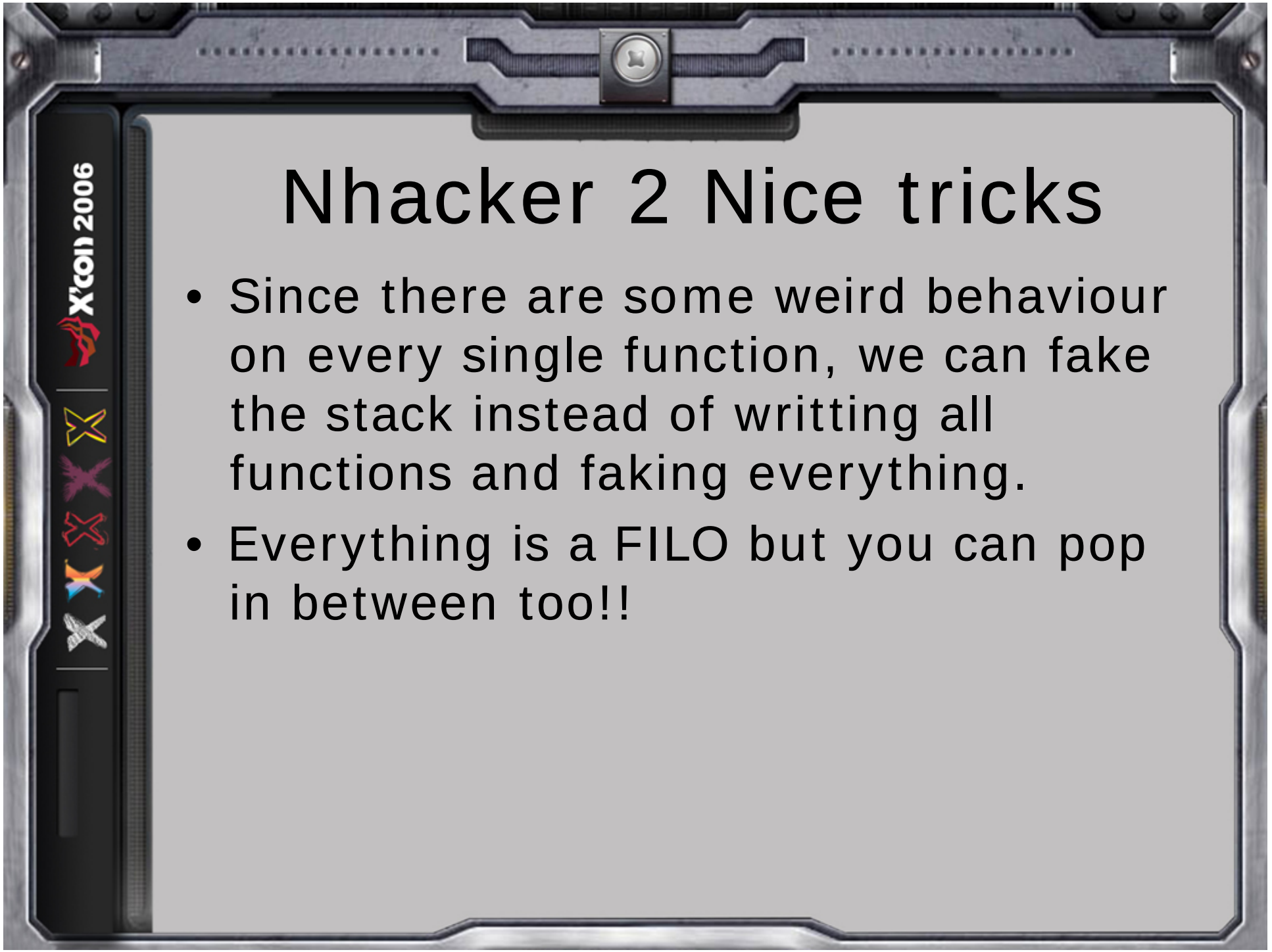
Nhacker 2.0 Structures

```
class parser {

public:

parser();

char * create_function_stack(char *);

int check_functions(char *);

int read_line(FILE *);

void decompose_line();

void debug(char *);

char ** get_argument_strcpy(char *);

char ** get_argument_memcpy(char *);

char ** get_argument_printf(char *);
```



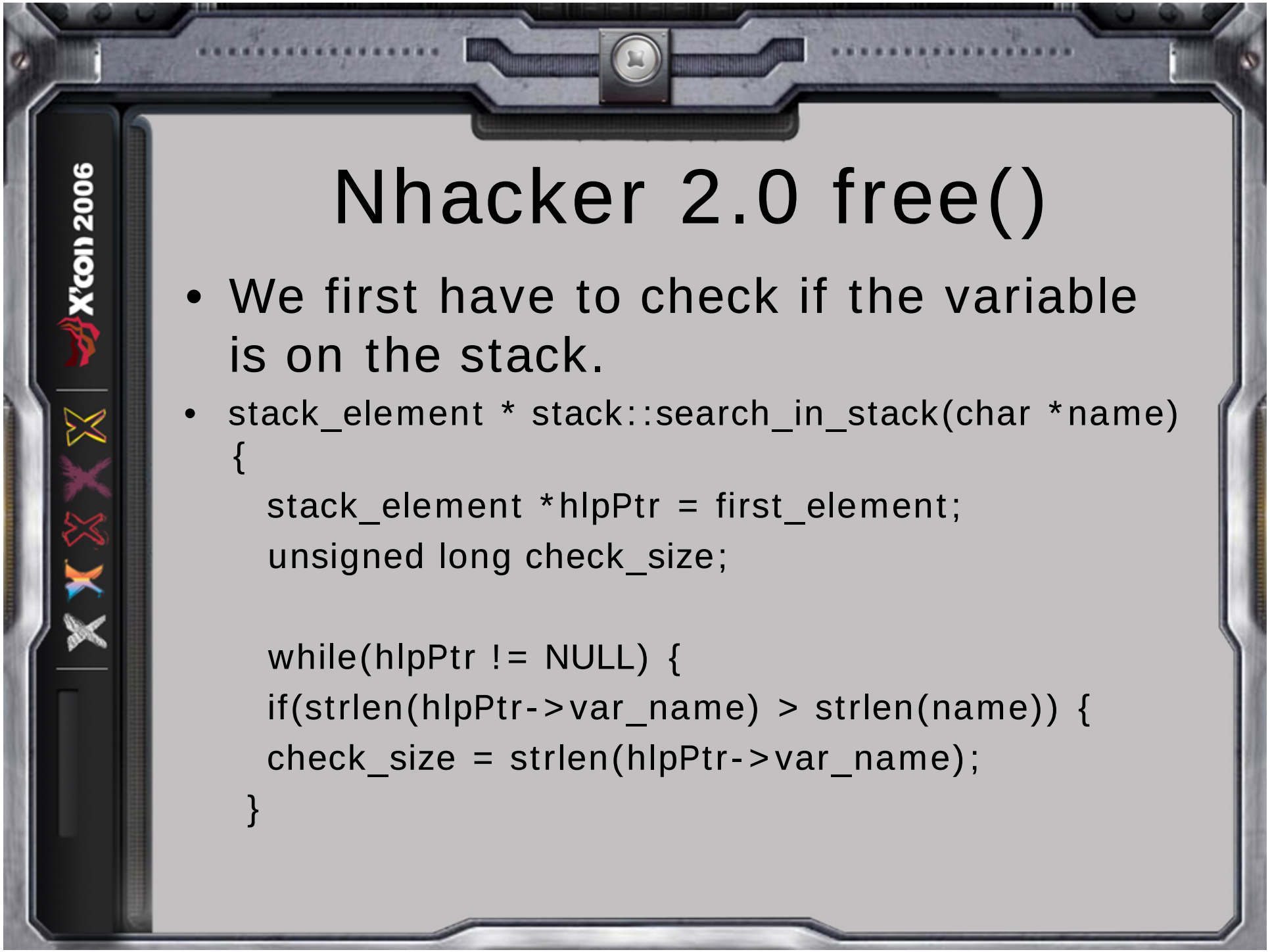
Nhacker 2.0 Structures

```
char ** get_variables_from_line(char *, unsigned long, int);
.....
private:
char *name;
char line[1024];
char *dangerous_functions[4];
};
```



Nhacker 2 Nice tricks

- Since there are some weird behaviour on every single function, we can fake the stack instead of writting all functions and faking everything.
- Everything is a FILO but you can pop in between too!!



Nhacker 2.0 free()

- We first have to check if the variable is on the stack.
- stack_element * stack::search_in_stack(char *name)
  {
    stack_element *hlpPtr = first_element;
    unsigned long check_size;

    while(hlpPtr != NULL) {
    if(strlen(hlpPtr->var_name) > strlen(name)) {
    check_size = strlen(hlpPtr->var_name);
   }

Nhacker 2.0 free()

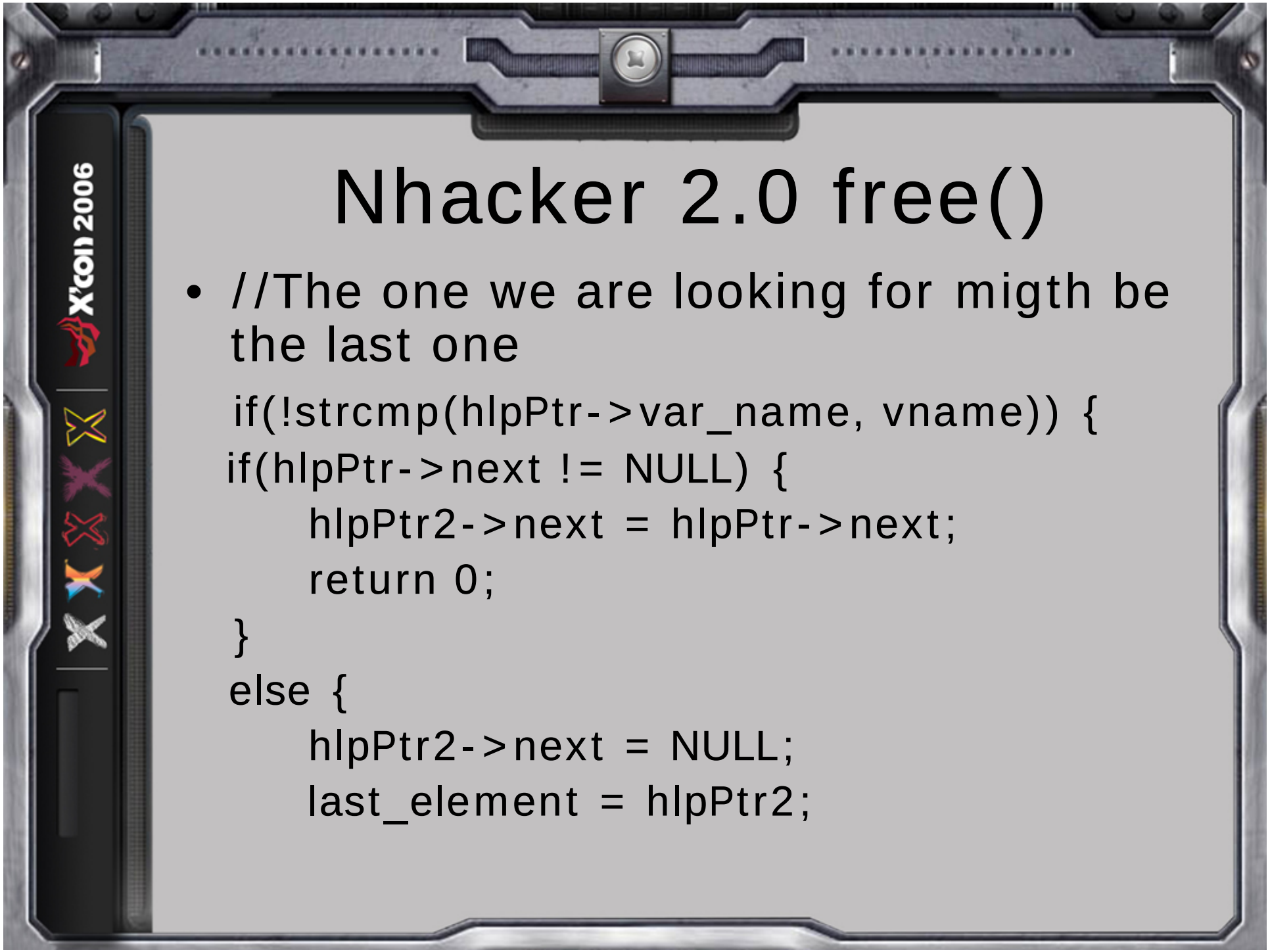
- else {
  check_size = strlen(name);
  }

```
if(!(strncmp(hlpPtr->var_name, name, check_size))) {
      return hlpPtr;
 }
     hlpPtr = hlpPtr->next;
}
return NULL;
}
```

Nhacker 2.0 free()

- We now (if the variable was on the stack) “free¡” the variable.

```
int stack::pop_stack(char *vname){
stack_element *hlpPtr = first_element;
stack_element *hlpPtr2 = hlpPtr;

while((hlpPtr->var_name != vname) || (hlpPtr-
 >next != NULL)) {
hlpPtr2 = hlpPtr;
hlpPtr=hlpPtr->next;
}
```



Nhacker 2.0 free()

- //The one we are looking for migth be the last one

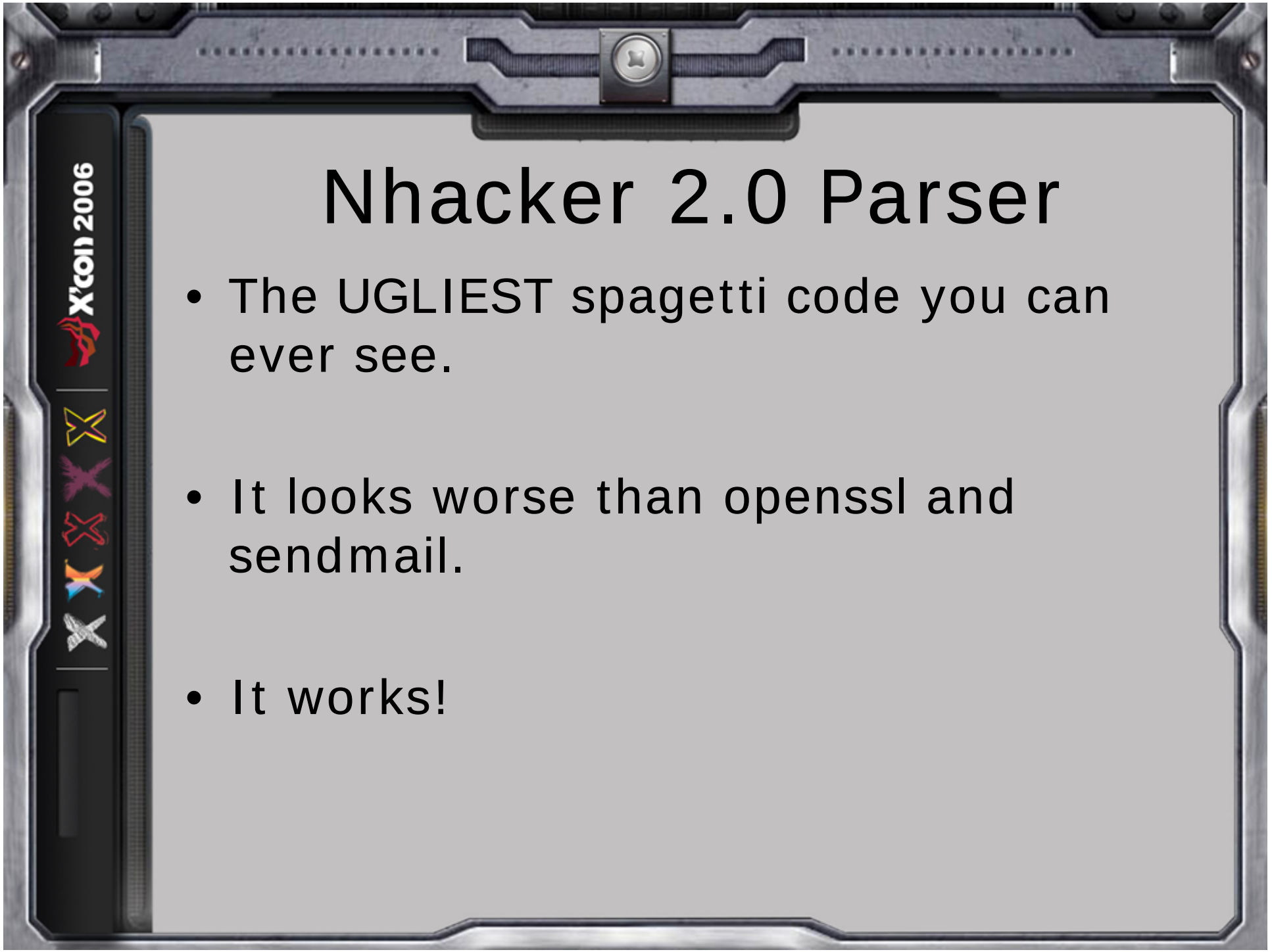
```
if(!strcmp(hlpPtr->var_name, vname)) {
if(hlpPtr->next != NULL) {
    hlpPtr2->next = hlpPtr->next;
    return 0;
}
else {
    hlpPtr2->next = NULL;
    last_element = hlpPtr2;
```

Nhacker 2.0 free()

```
            free(hlpPtr);
        return 0;
}
}
        return -1;
}
```

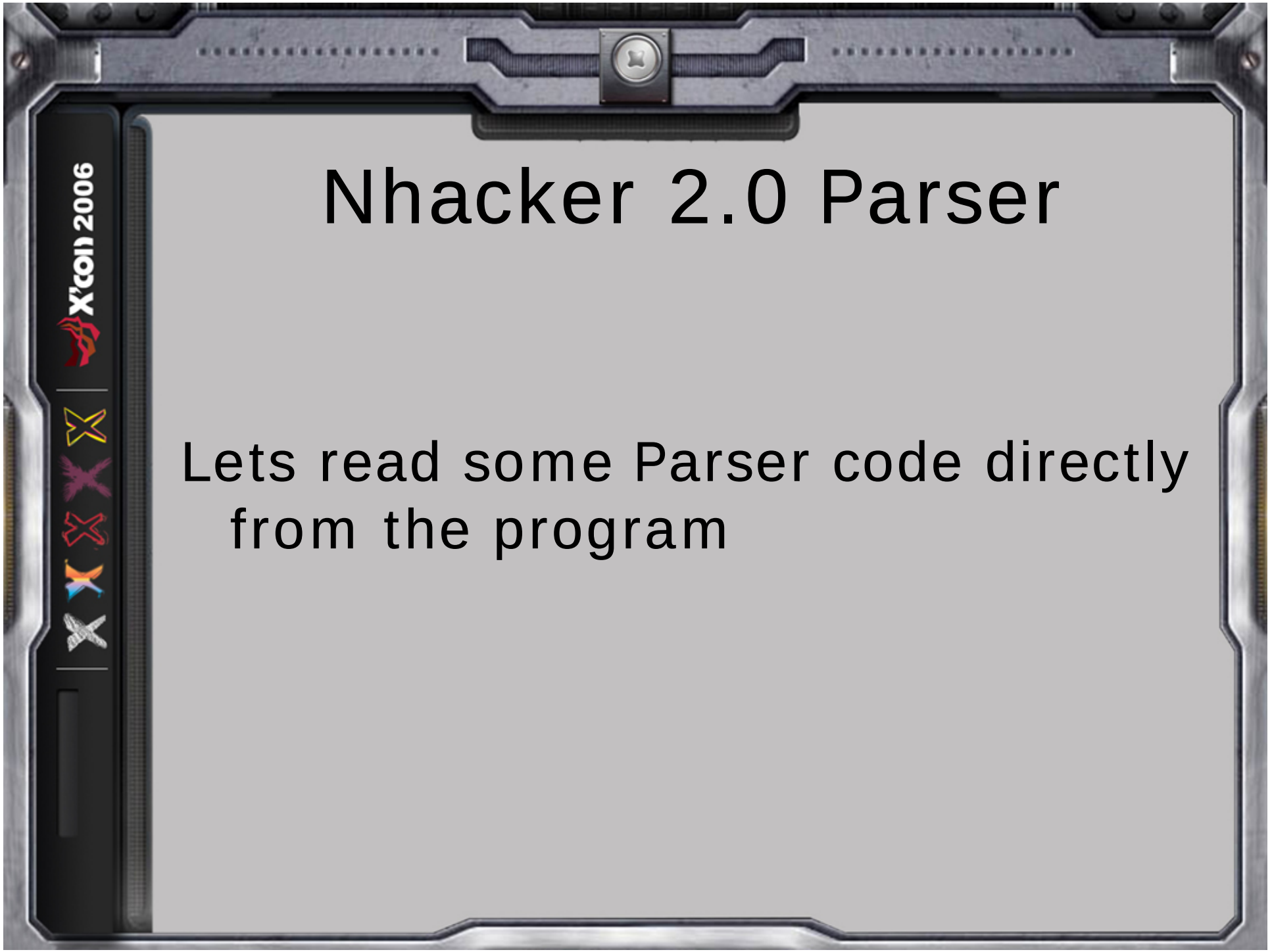
Nhacker 2.0 free()

- Checking if the variable is on the stacklet us know if we have a double free, and lets us check if we have a memory leak.
- Popping from the stack we can also have a good idea on how the stack at any given time.



Nhacker 2.0 Parser

- The UGLIEST spagetti code you can ever see.
- It looks worse than openssl and sendmail.
- It works!

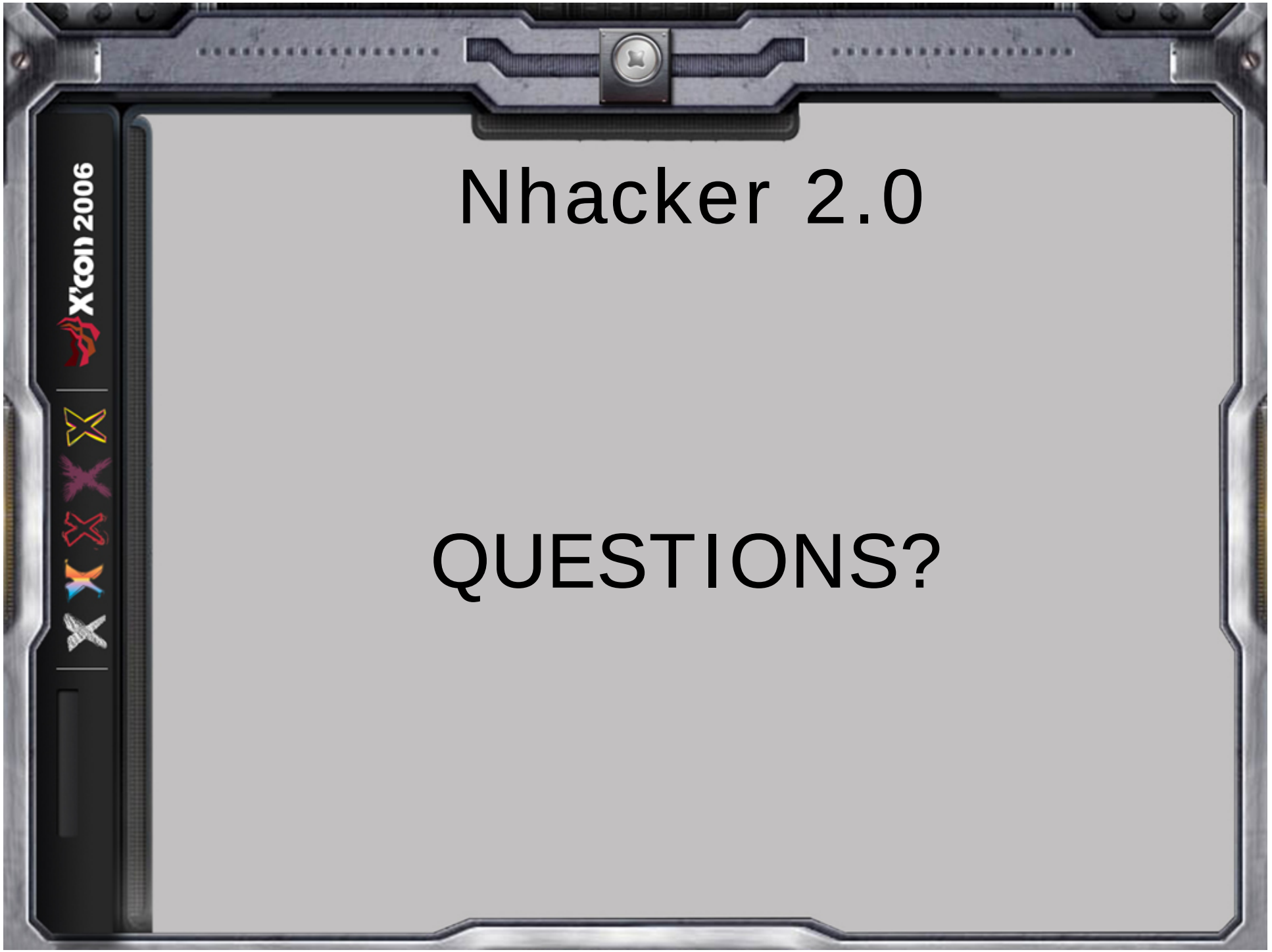


Nhacker 2.0 Parser

Lets read some Parser code directly from the program

Nhacker 2.0 Examples

- Real Life Examples on Nhacker finding bugs.
  - C Example bug.
  - C Remote bug.
  - C 0hday bug.


X'con 2006
Nhacker 2.0
QUESTIONS?

THANK YOU!

- I want to thank Casper for putting up with my bad english.
- Fyodor Yarochkin for being such an awsome friend.
- Ana Laura for being my sunshine.
- God for being with me all this years.
- And you for listening patiently!