

Uninitialized Variables

Finding, Exploiting, Automating

I am

- ▶ **Daniel Hodson**

- ▶ Work for a company, breaking things.
- ▶ Involved in several online communities/groups like overthewire.org, felinemenace, LUMC thug
- ▶ Presented at Ruxcon 2004/2006.

Definition

Wikipedia

“In computing, an uninitialized variable is a variable that is declared but is not set to a definite known value before it is used.”



Agenda

- ▶ **Introduction**

- ▶ Stack Layout demo
- ▶ Heap Layout demo

- ▶ **Exploiting**

- ▶ Protection mechanisms
- ▶ Methodologies

- ▶ **Question/Answer**

- ▶ **Finding**

- ▶ Case studies
- ▶ Code demonstrations

- ▶ **Automating**

- ▶ Compile time
- ▶ Run time



Introduction

History

- ▶ Some good sources are eEye, Shellcoders Handbook 2nd edition, TAOSSA, and Cansecwest slides.



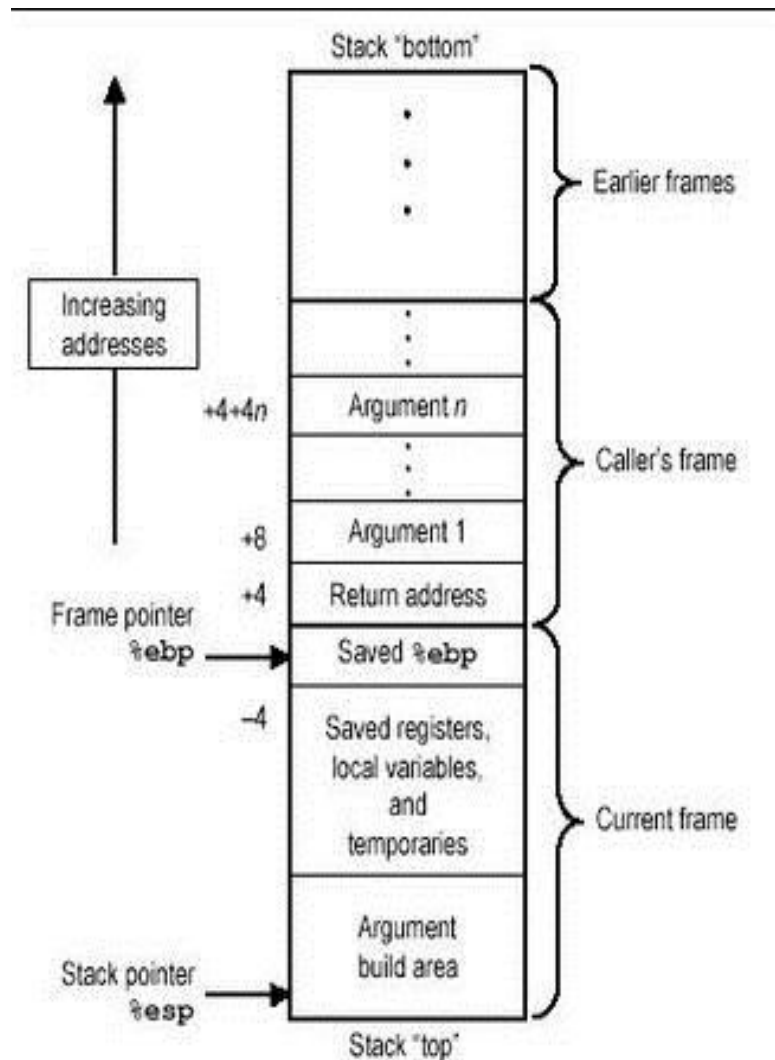
Stack Introduction

C99 standard

“The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.”.



Stack



Stack example

```
void function_one(int arg1, int arg2)
{
    long local1 = arg1;
    long local2 = arg2;
    printf("%d\n", local1);
    printf("%d\n", local2);
    return;
}

void function_two(int arg1, int arg2)
{
    long local1_uninitialized;
    long local2_uninitialized;
    printf("%d\n", local1_uninitialized);
    printf("%d\n", local2_uninitialized);
    return;
}

int main(int argc, char **argv)
{
    function_one(1, 2);
    function_two(3, 4);
}
```



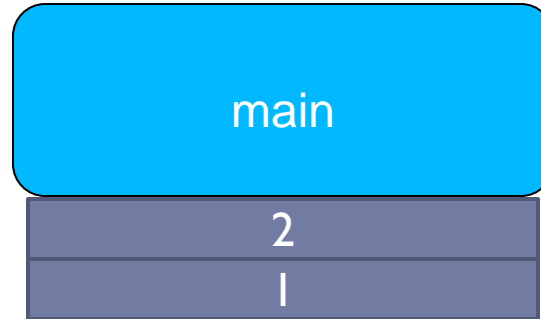
Stack Layout



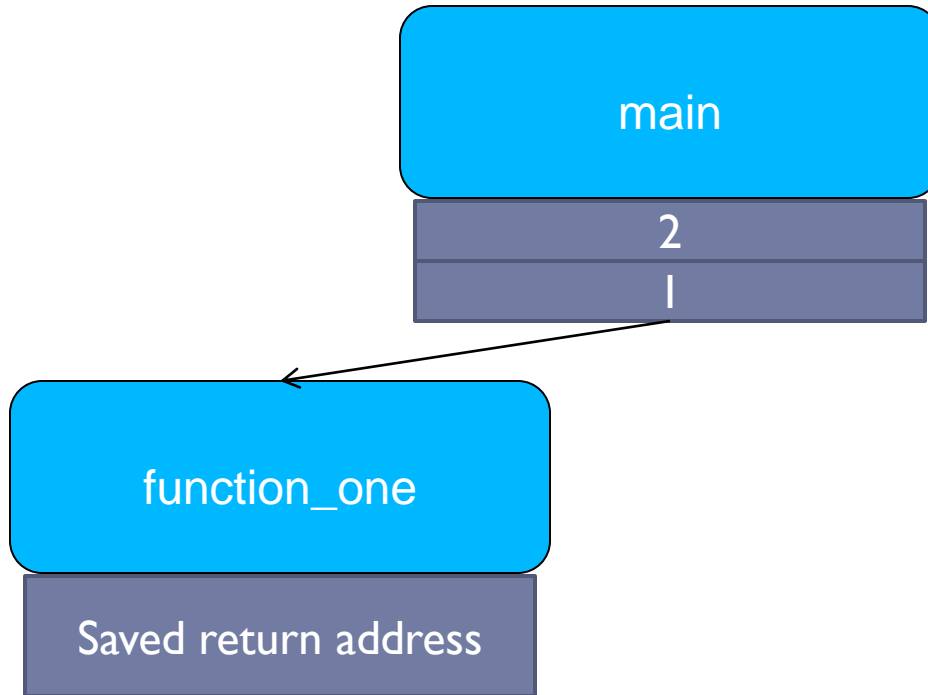
Stack Layout



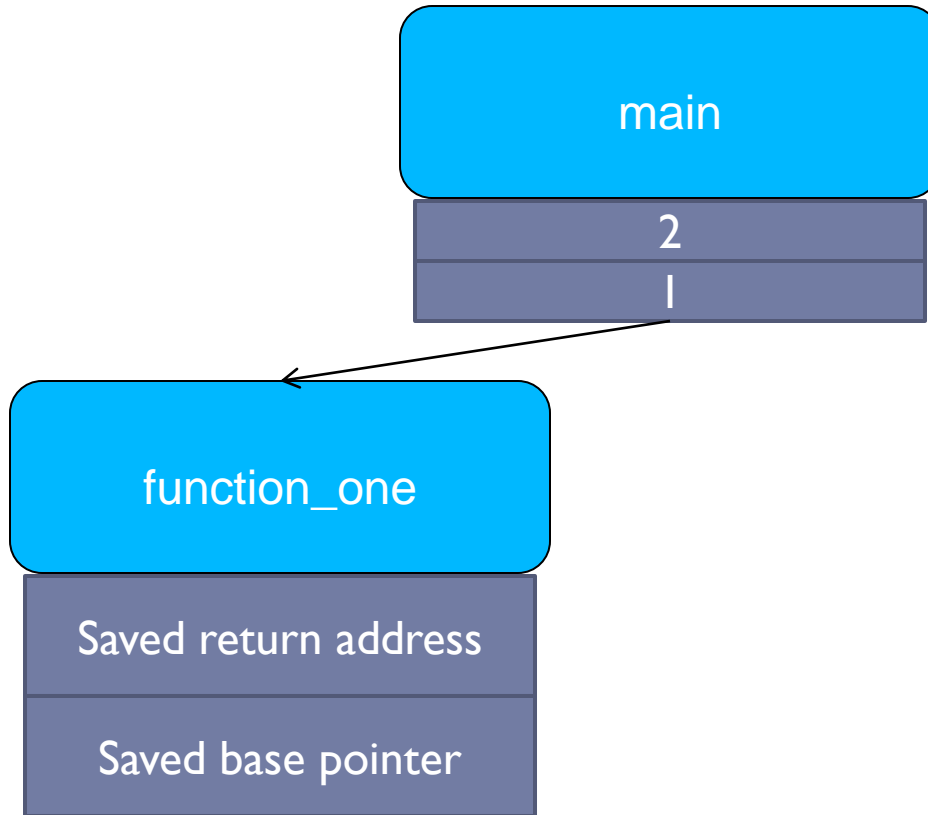
Stack Layout



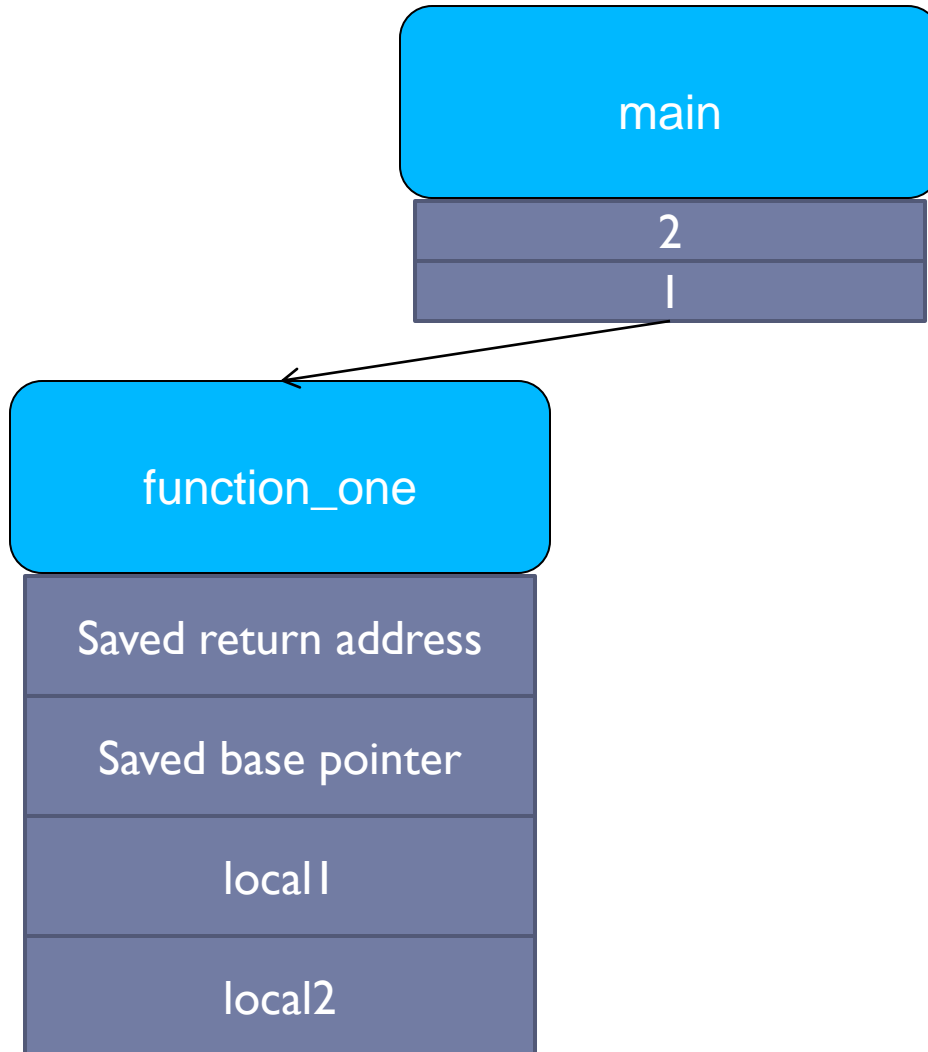
Stack Layout



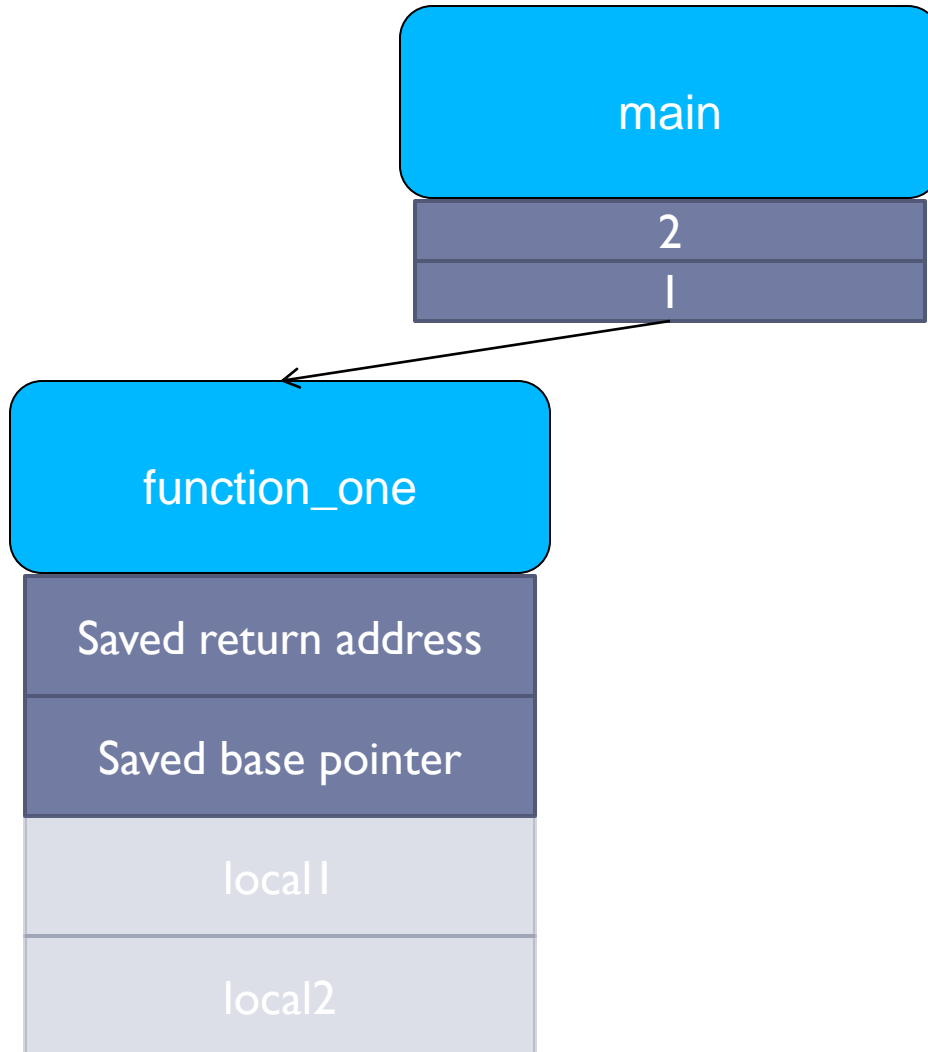
Stack Layout



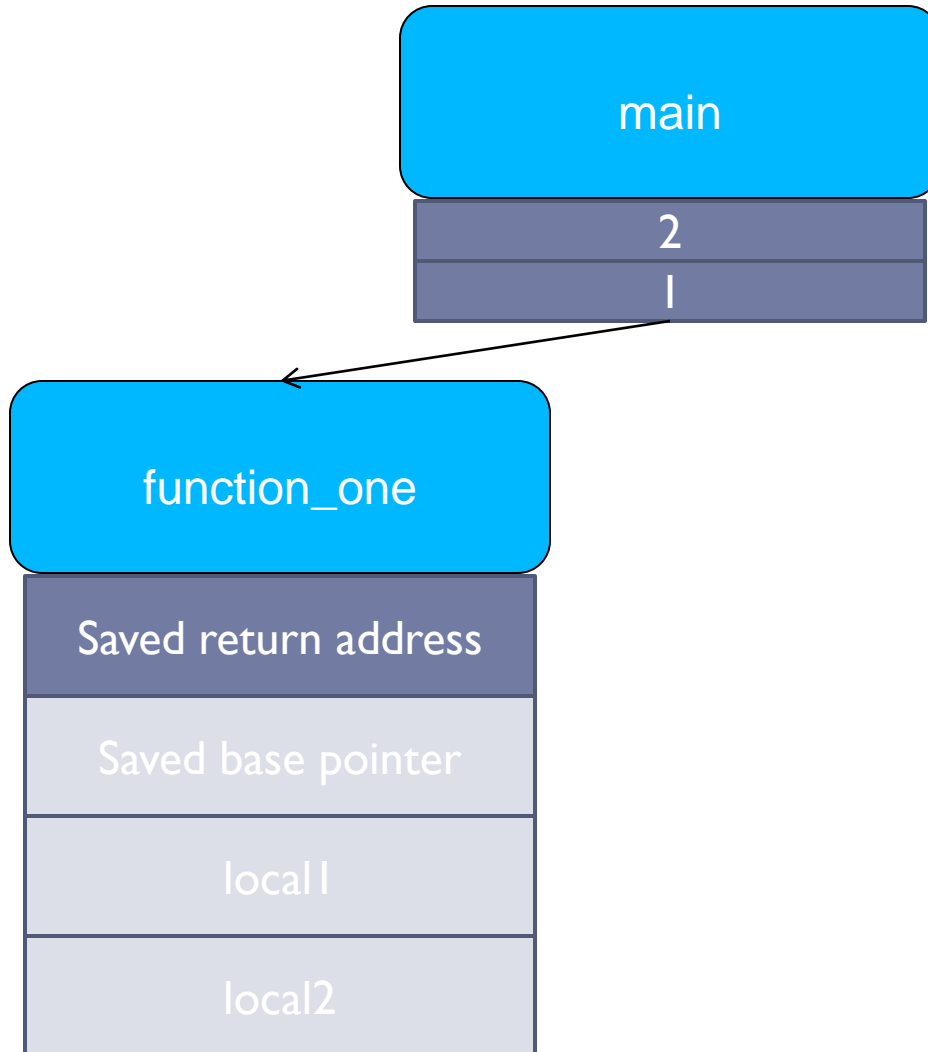
Stack Layout



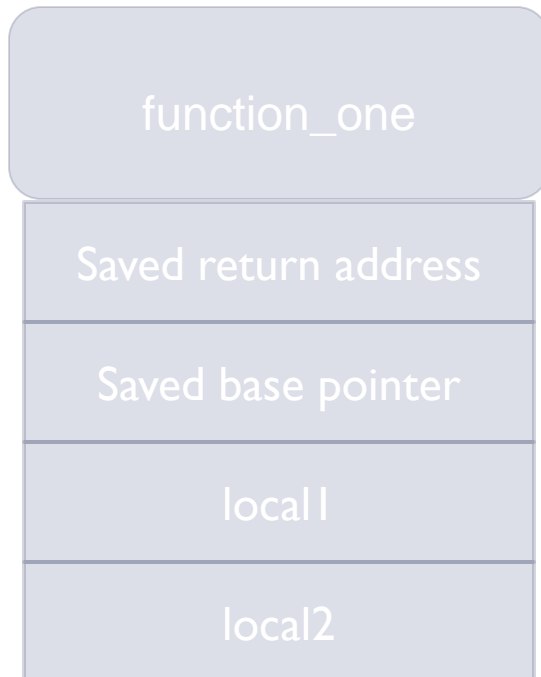
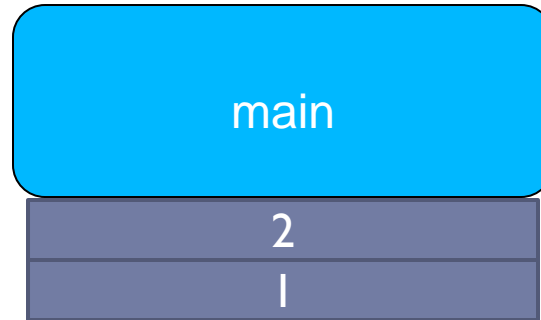
Stack Layout



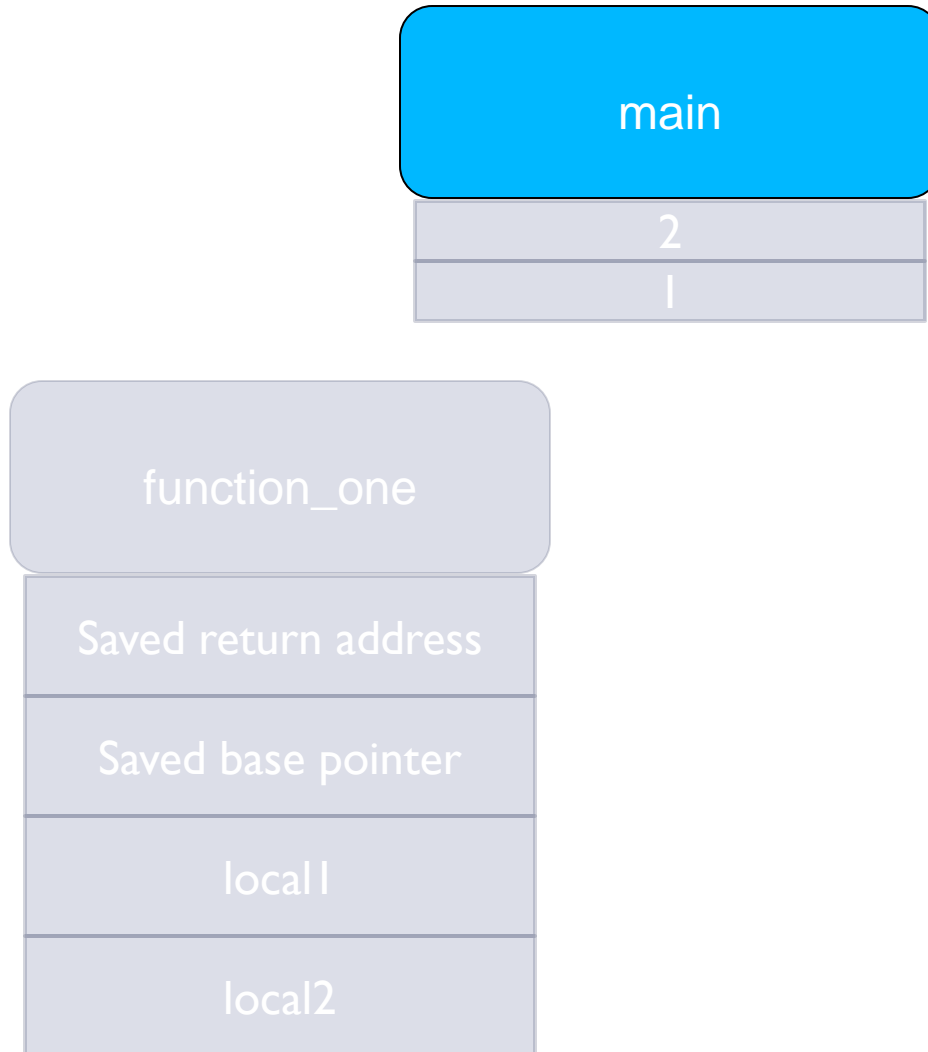
Stack Layout



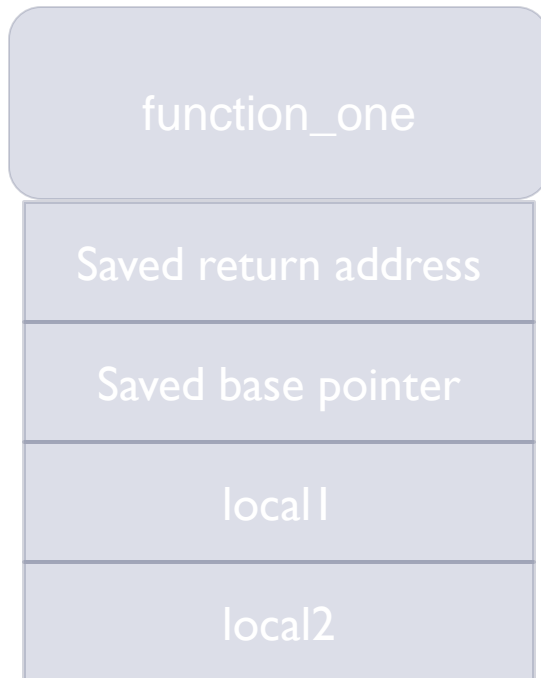
Stack Layout



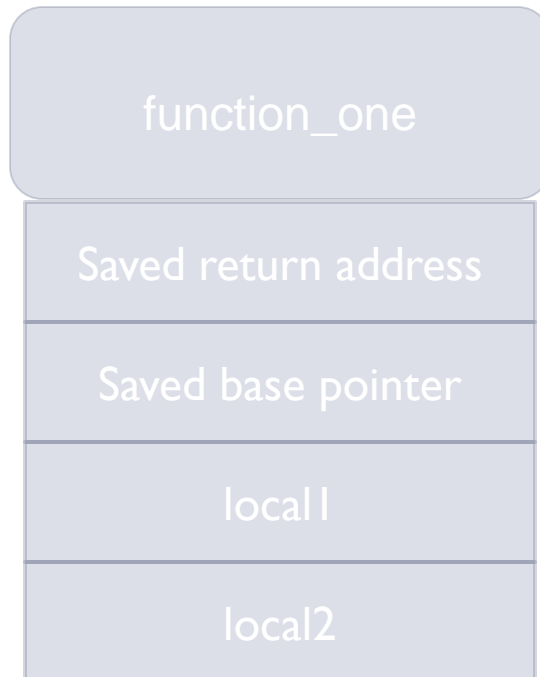
Stack Layout



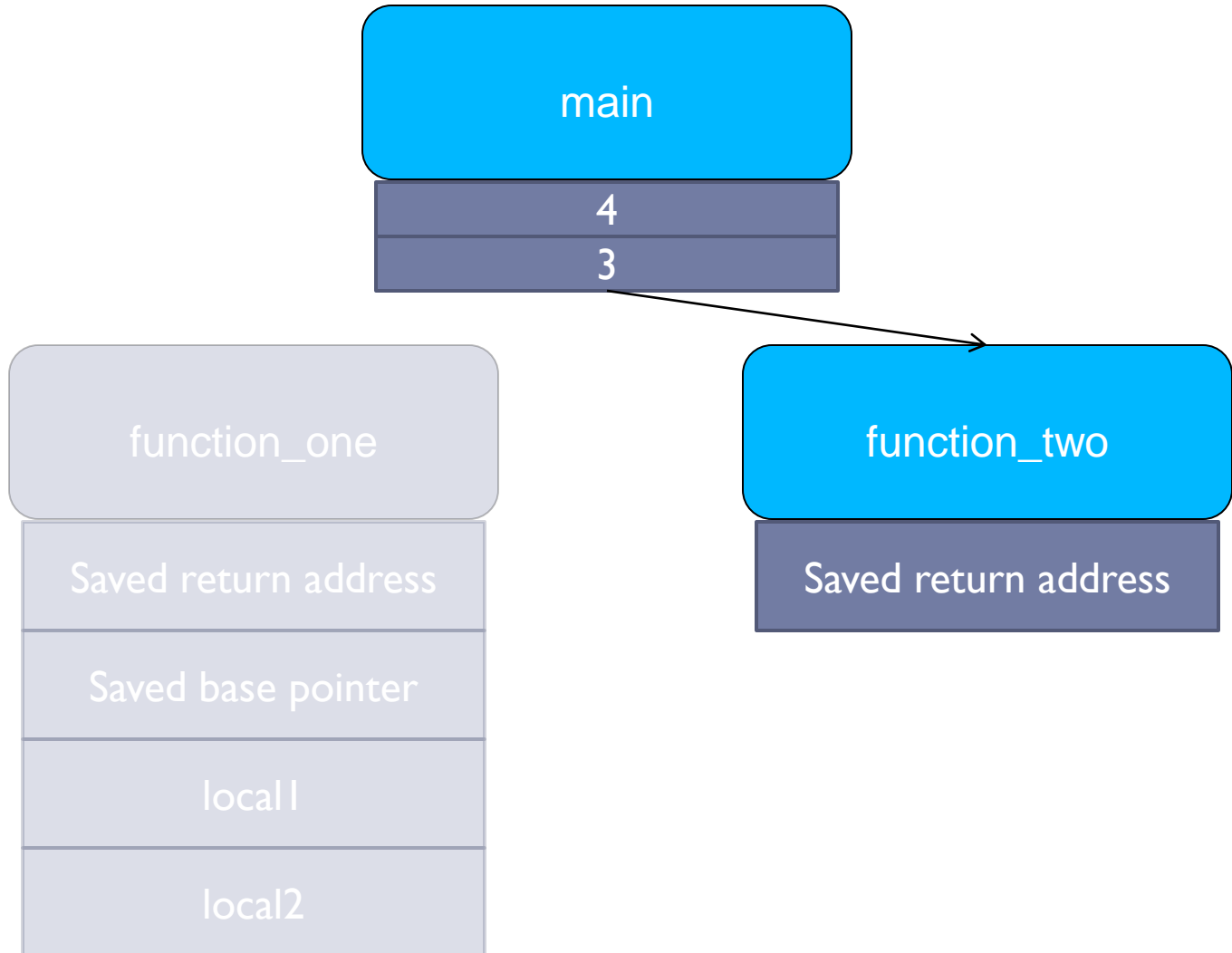
Stack Layout



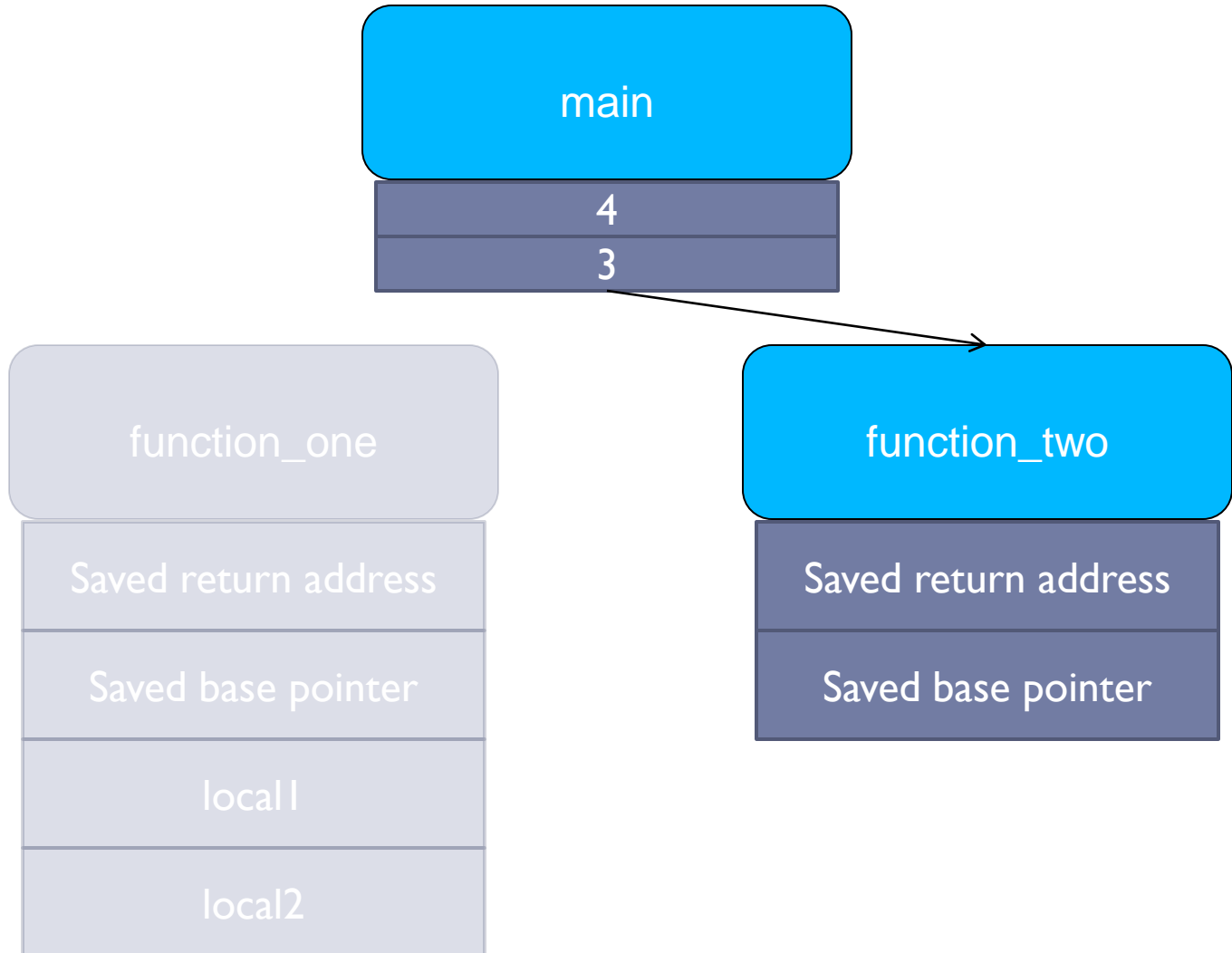
Stack Layout



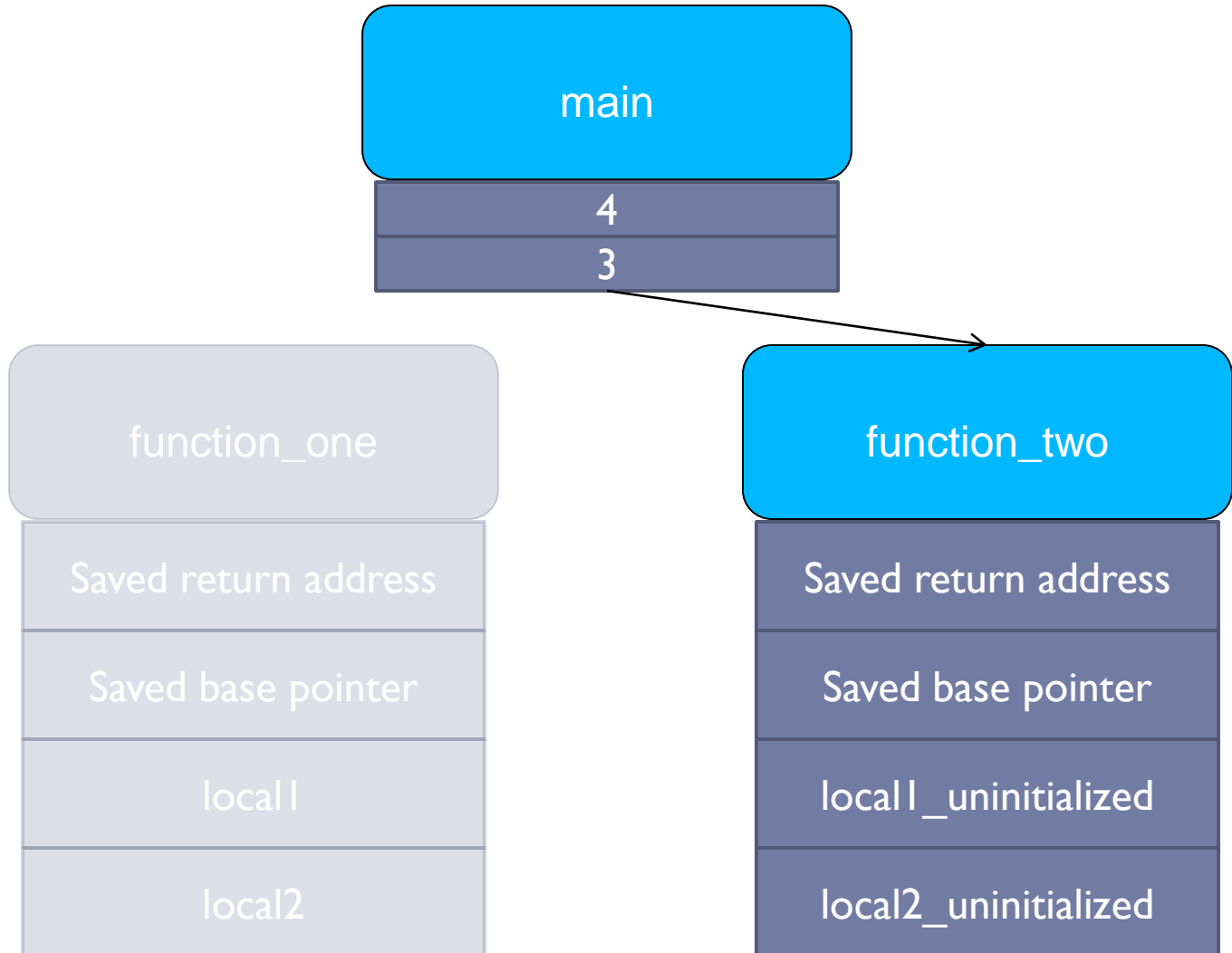
Stack Layout



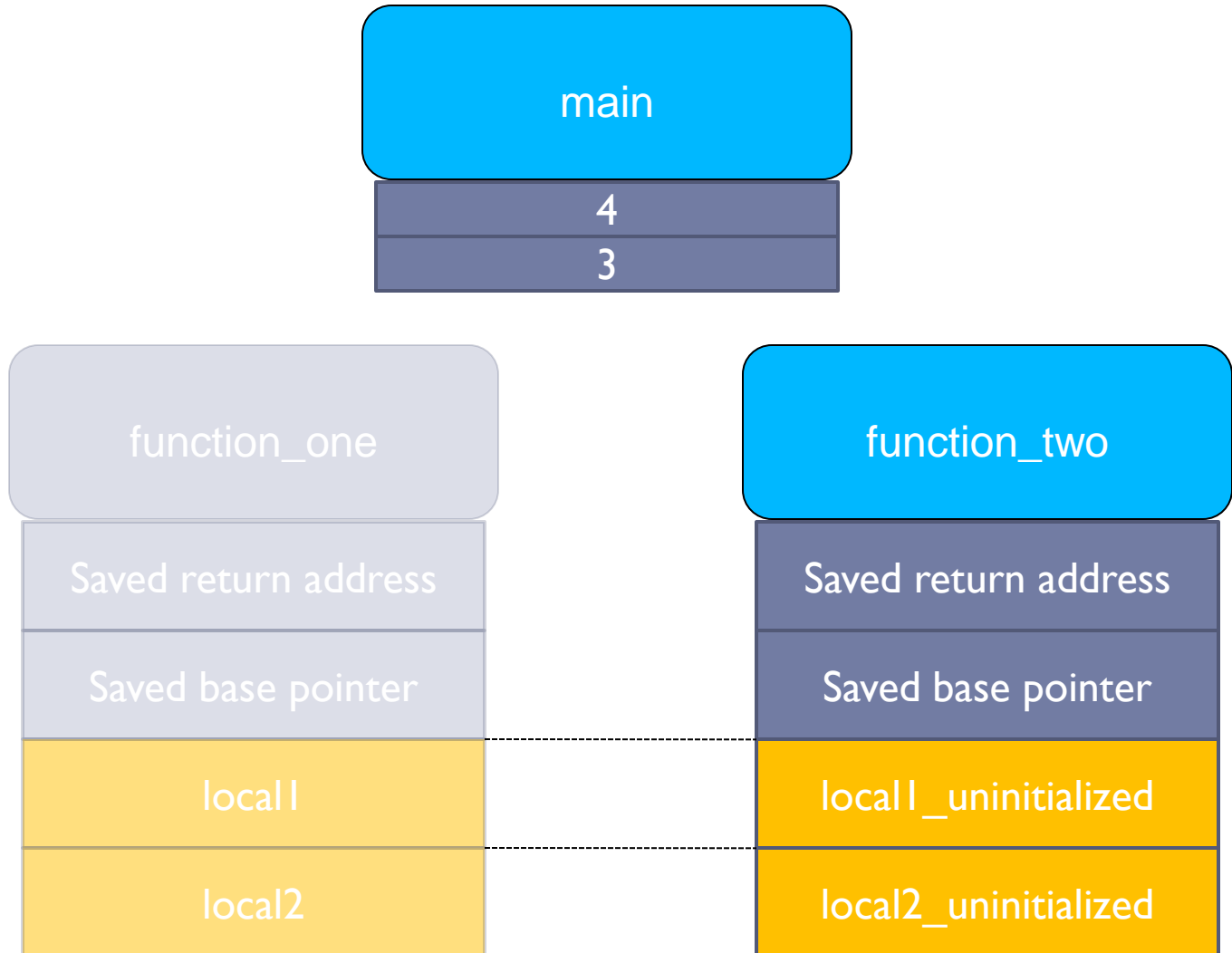
Stack Layout



Stack Layout



Stack Alignment

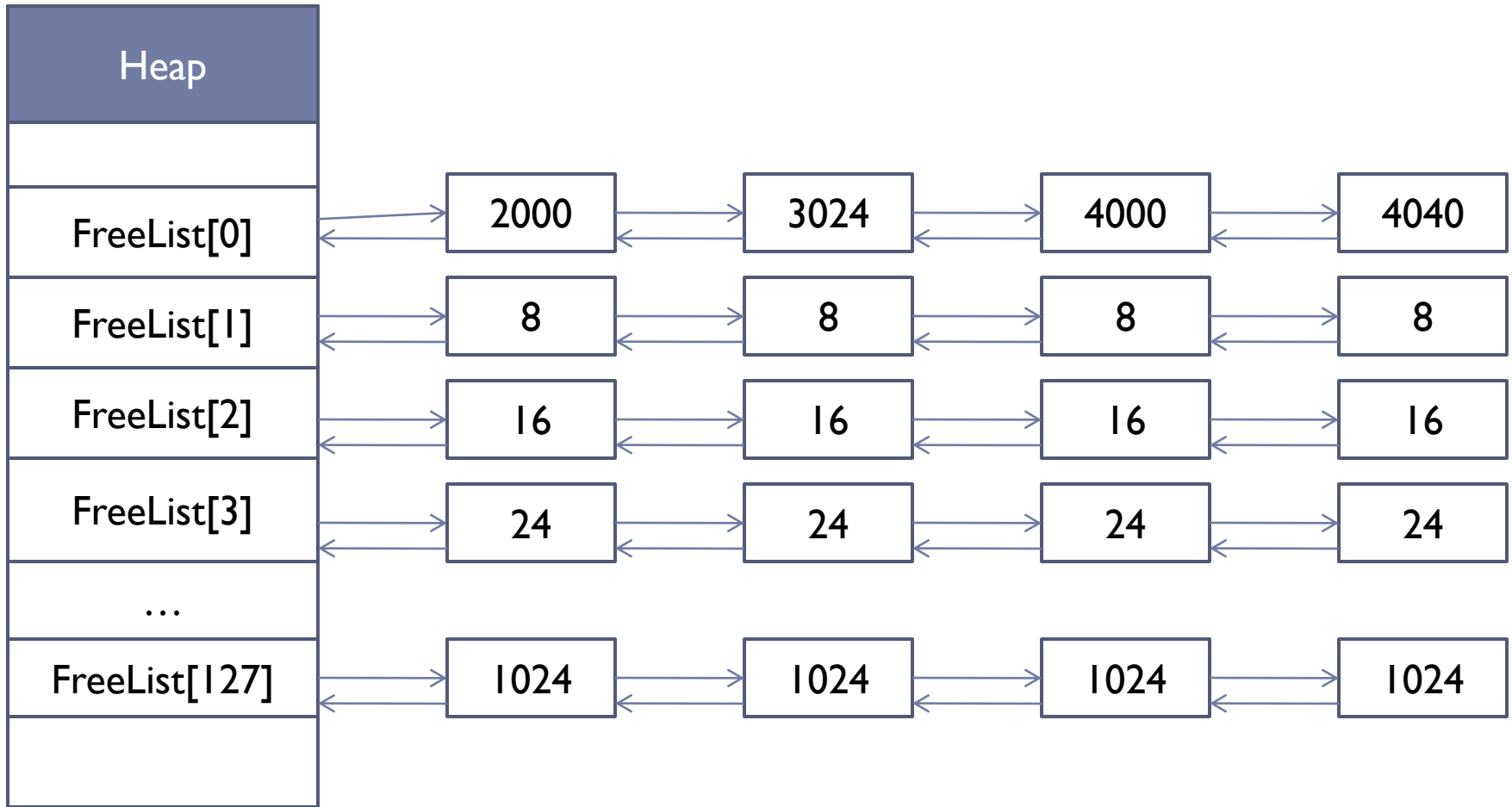


Heap Introduction

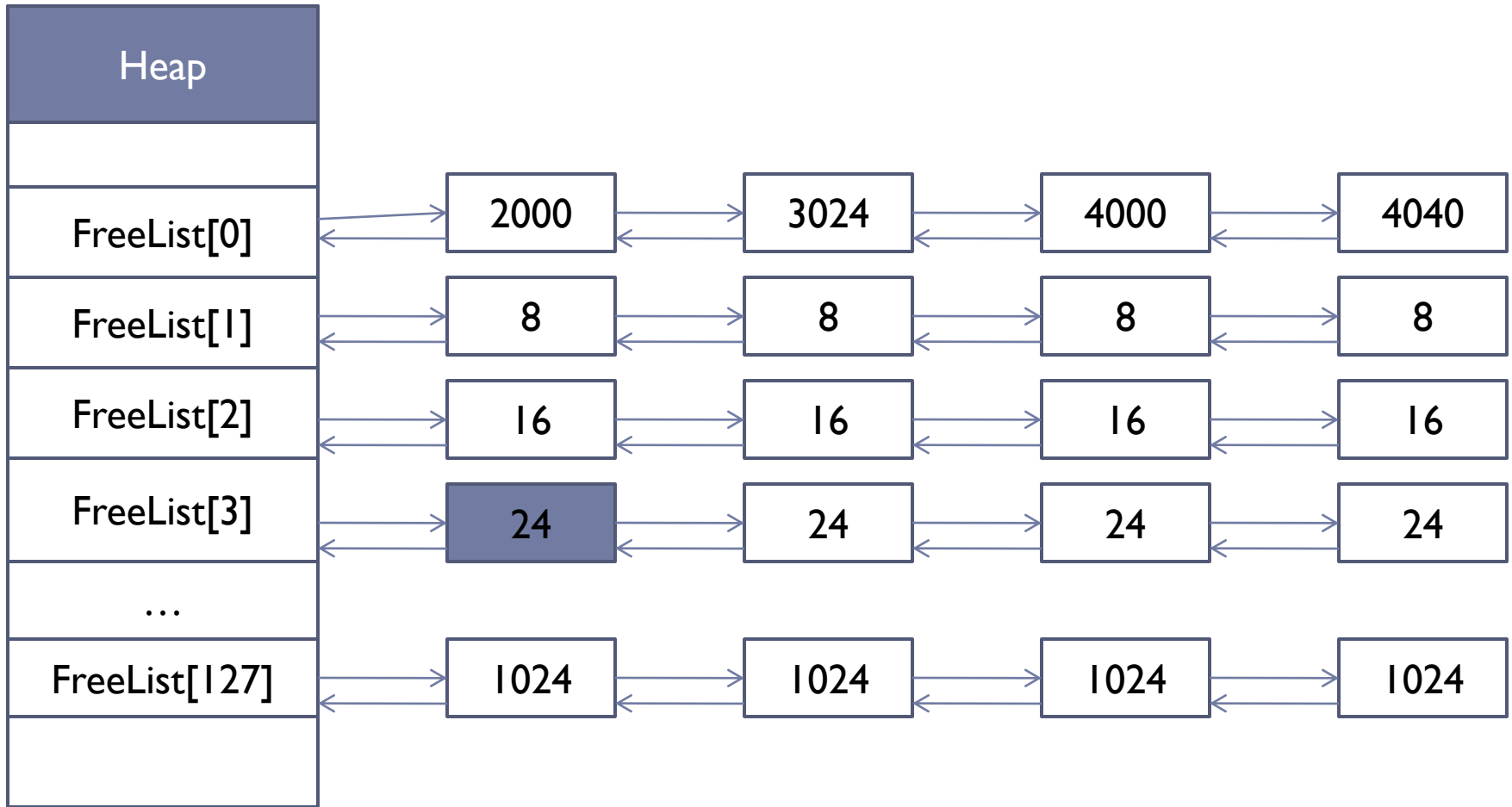
- ▶ Sometimes a variable will need dynamic storage. This storage can be requested by the memory allocator at runtime.
- ▶ The memory allocator will allocate the requested storage size, and return a chunk of memory back to the variable.
- ▶ This memory can sometimes contain residual data from previous allocations.



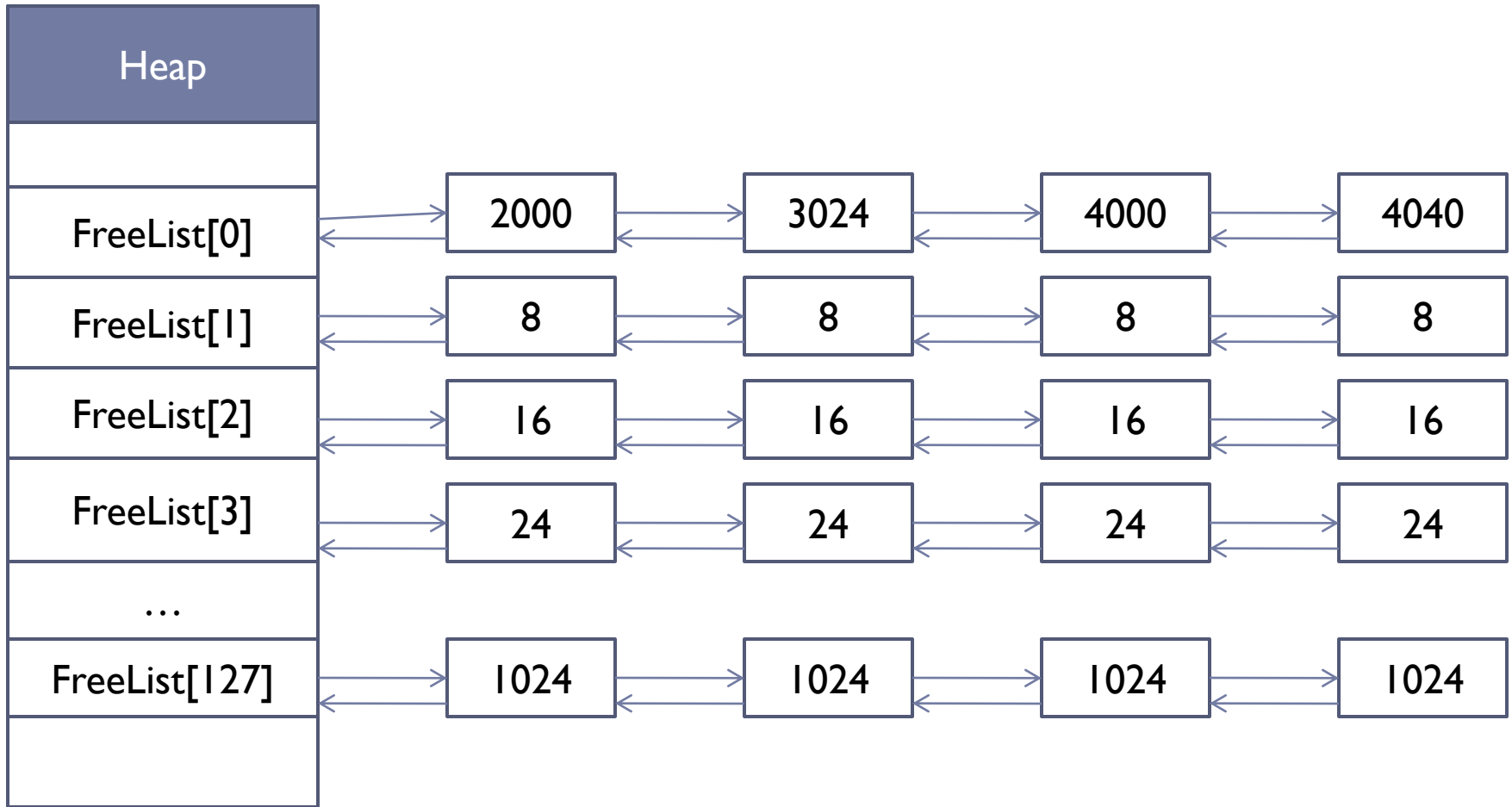
Heap Layout



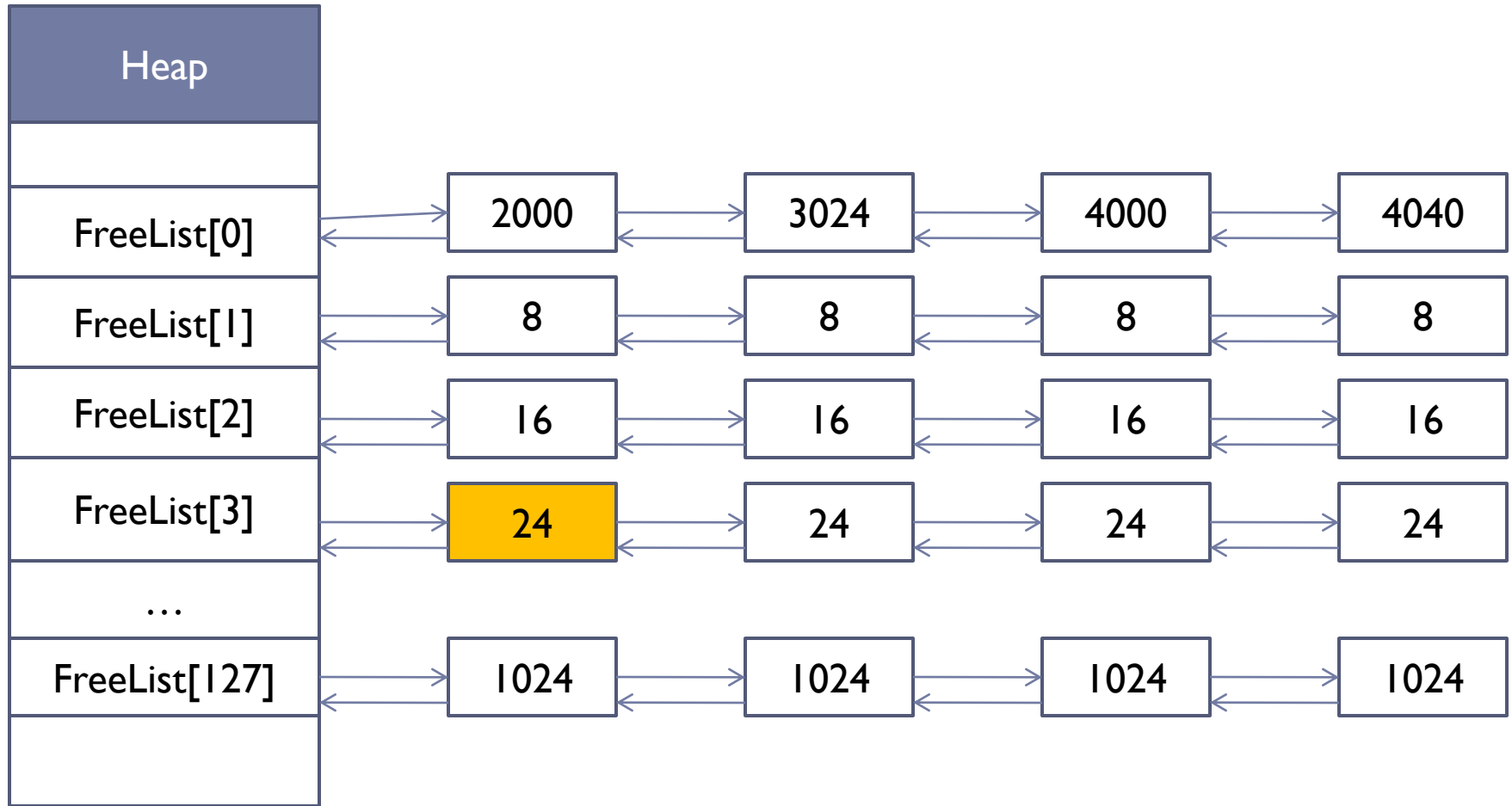
Heap Layout



Heap Layout



Heap Layout





Finding

What we'll be covering

- ▶ Local variables
- ▶ Unchecked returns
- ▶ Branch instructions
- ▶ Array initialization/termination
- ▶ Structure padding
- ▶ Uninitialized memory
- ▶ Use after free
- ▶ Class ctor/dtor
- ▶ Threads
- ▶ Return expressions



Local automatic variables

- ▶ Local variables are usually declared at the beginning of a function. Any variables that aren't initialized at this time are considered “uninitialized”.
- ▶ Sometimes, they are left uninitialized because the programmer doesn't know their value until runtime.
- ▶ These are the variables of most interest.
- ▶ <http://codesearch.google.com> is your friend 😊



Local automatic Variables: *case study*

```
typedef struct mixer_info
{
    char id[16];
    char name[32];
    int  modify_counter;
    int  fillers[10];
} mixer_info;
```

Local automatic Variables: *case study*

```
@@ -2016,6 +2018,8 @@ static int usb_audio_ioctl_mixdev(struct
if (cmd == SOUND_MIXER_INFO)
{
    mixer_info info;

    strncpy(info.id, "USB_AUDIO", sizeof(info.id));
    strncpy(info.name, "USB Audio Class Driver",
sizeof(info.name));
    info.modify_counter = ms->modcnt;

    if (copy_to_user((void __user *)arg, &info,
sizeof(info)))
        return -EFAULT;

return 0;
```

Local automatic Variables: *case study*

- ▶ The 'fillers' array is left uninitialized.
- ▶ The `copy_to_user` will copy the entire structures contents back to user space.
- ▶ This will leak 40 bytes of kernel information.

```
typedef struct mixer_info
{
    char id[16];
    char name[32];
    int  modify_counter;
    int fillers[10];
} mixer_info;
```

Local automatic Variables: *case study*

```
@@ -2016,6 +2018,8 @@ static int usb_audio_ioctl_mixdev(struct
if (cmd == SOUND_MIXER_INFO)
{
    mixer_info info;

+ memset(&info, 0, sizeof(info));

    strncpy(info.id, "USB_AUDIO", sizeof(info.id));
    strncpy(info.name, "USB Audio Class Driver", sizeof(info.name));
    info.modify_counter = ms->modcnt;

    if (copy_to_user((void __user *)arg, &info, sizeof(info)))
        return -EFAULT;

    return 0;
```

Initialization Functions: *return values*

- ▶ Variables can be initialized via a function call.
- ▶ Sometimes the function may fail, and return an error value to indicate the problem.
- ▶ It's up to the programmer to check the return value, otherwise the variable may be left uninitialized.
- ▶ A good example of a family of functions that are vulnerable to this is the `*scanf` functions.



Initialization functions: *return values* *example*

```
int main(void)
{
    int age;
    fscanf(stdin, "%d", &age);
    printf("You are %d years old.\n", age);
    return 0;
}
```



Initialization functions: *return values* *example*

```
int main(void)
{
    int age;
    if(fscanf(stdin, "%d", &age) == NULL) return
        -1;
    printf("You are %d years old.\n", age);
    return 0;
}
```



Branch instructions:

- Conditional branch instructions responsible for initializing variables could be vulnerable.
 - If, else
 - For
 - While, do while
 - Switch
- The code must account for cases in which a condition is NOT met.



Branch Instructions: *switch case study*

```
int off = *offp, rhlen;
...
switch (rh->ip6r_type) {
case IPV6_RTHDR_TYPE_0:
    if (rh->ip6r_segleft == 0)
        break; /* Final dst. Just ignore the header. */
    rhlen = (rh->ip6r_len + 1) << 3;
    /*
    * note on option length:
    * maximum rhlen: 2048
    ...}

*offp += rhlen;
return (rh->ip6r_nxt);
}
```

Branch Instructions: *switch case study*

```
int off = *offp, rhlen;
...
    switch (rh->ip6r_type) {
    case IPV6_RTHDR_TYPE_0:
+       rhlen = (rh->ip6r_len + 1) << 3;
        if (rh->ip6r_segleft == 0)
            break; /* Final dst. Just ignore the header. */
        /*
         * note on option length:
         * maximum rhlen: 2048
        */
    ...}

*offp += rhlen;
return (rh->ip6r_nxt);
}
```

Array initialization and termination

- ▶ Arrays are a data structure which consists of a group of elements of the same data type, and are accessed through an index.
- ▶ It's not uncommon for code to only partially initialize an array, leaving the other half uninitialized.



Array initialization and termination: *example*

```
void recvloop()  
{  
    char buffer[256];  
  
    read(STDIN_FILENO, &buffer, sizeof(buffer));  
    buffer[sizeof(buffer) - 1] = '\0';  
  
    printf("%s\n", buffer);  
}
```



Array initialization and termination: *example*

```
void recvloop()  
{  
    char buffer[256] = {0};  
  
    read(STDIN_FILENO, &buffer, sizeof(buffer));  
    buffer[sizeof(buffer) - 1] = '\\0';  
  
    printf("%s\\n", buffer);  
}
```



Structure padding

- ▶ For alignment purposes, sometimes structures will contain “padding” data between local variables.
- ▶ It's not possible to initialize this padding data by any structure member, so initialization must be done with `memset`.



Structure padding: *example*

```
typedef struct
{
    char local1;
    long local2;
} STRUC;

int main(void)
{
    STRUC structure;

    printf("%d\n", sizeof(structure.local1));
    printf("%d\n", sizeof(structure.local2));
    printf("%d\n", sizeof(structure));

    return 0;
}
```



Memory: *heap*

- ▶ Sometimes local variables require dynamic storage which can only be computed at runtime.
- ▶ This dynamic storage is requested at runtime by calls to the heap allocator.
- ▶ *CERT EXP33-CPP*
 - “Additionally, memory allocated by functions such as `malloc()` should not be used before initialized as its contents are indeterminate.”



Memory: *heap case study*

```
typedef struct resource_record RESOURCE_RECORD_T;
struct resource_record
{
    char                *rr_domain;
    unsigned int        rr_type;
    unsigned int        rr_class;
    unsigned int        rr_ttl;
    unsigned int        rr_size;
    union
    {
        void            *rr_data;
        MX_RECORD_T     *rr_mx;
        MX_RECORD_T     *rr_afsdb; /* mx and afsdb are identical */
        SRV_RECORDT_T   *rr_srv;
        struct in_addr   *rr_a;
        char             *rr_txt;
    } rr_u;
    RESOURCE_RECORD_T *rr_next;
};
```

Memory: *heap case study*

```
static DNS_REPLY_T * parse_dns_reply(unsigned char *data, int len) {
...
    r = (DNS_REPLY_T *) sm_malloc(sizeof(*r));
    if (r == NULL) return NULL;
    memset(r, 0, sizeof(*r));
...
    rr = &r->dns_r_head;
    while(...) {
        ...
        if (p + size > data + len) {
            dns_free_data(r); return NULL;
        }
        *rr = (RESOURCE_RECORD_T *) sm_malloc(sizeof(**rr));
        if (*rr == NULL) {
            dns_free_data(r); return NULL;
        }
        (*rr)->rr_domain = sm_strdup(host);
...
        rr = &(*rr)->rr_next;
```

Memory: *heap case study*

```
void dns_free_data( DNS_REPLY_T *r)
{
    RESOURCE_RECORD_T *rr;
    if (r->dns_r_q.dns_q_domain != NULL)
        sm_free(r->dns_r_q.dns_q_domain);

    for (rr = r->dns_r_head; rr != NULL; )
    {
        RESOURCE_RECORD_T *tmp = rr;
        if (rr->rr_domain != NULL)
            sm_free(rr->rr_domain);
        if (rr->rr_u.rr_data != NULL)
            sm_free(rr->rr_u.rr_data);
        rr = rr->rr_next;
        sm_free(tmp);
    }
    sm_free(r);
}
```

▶

Memory: *heap case study*

```
static DNS_REPLY_T * parse_dns_reply(unsigned char *data, int len) {
...
while(...)
{
    *rr = (RESOURCE_RECORD_T *) sm_malloc(sizeof(**rr));
    if (*rr == NULL) {
        dns_free_data(r);
        return NULL;
    }
+ memset(*rr, 0, sizeof(**rr));
    (*rr)->rr_domain = sm_strdup(host);
    if ((*rr)->rr_domain == NULL) {
        dns_free_data(r);
        return NULL;
    }
...
    rr = &(*rr)->rr_next;
}
}
```

Use after free

- ▶ A variable requests a chunk of memory from the allocator.
- ▶ That variable is then free'd.
- ▶ The variable continues to be used, manipulating and working with memory that has become “uninitialized”.



Classes

- ▶ C++ classes can contain constructor and destructor routines.
- ▶ Constructors should be audited for any variables that don't get initialized.
- ▶ Destructors should be audited for use after free as well as uninitialized variable vulnerabilities.



Classes: *example*

```
class Object
{
    public:
    char *initialized;
    char *uninitialized;

    Object()
    {
        this->initialized = (char *)calloc(1, 100);
    }
    ~Object()
    {
        free(this->initialized);
        free(this->uninitialized);
    }
};
```



Classes: *example*

```
class Object
{
    public:
    char *initialized;
    char *uninitialized;

    Object()
    {
        this->initialized = (char *)calloc(1, 100);
        this->uninitialized = (char *)calloc(1, 100);
    }
    ~Object()
    {
        free(this->initialized);
        free(this->uninitialized);
    }
};
```



Threads

- ▶ Multiple threads may request access to a shared resource by way of acquiring a mutex.
- ▶ Threads may execute at any point (not necessarily in the order they were created).
- ▶ If a thread makes an assumption of a shared resources state, it's possible uninitialized memory or variables will be accessed.



Threads: *example*

```
void *thread_one(void *a)
```

```
{
```

```
    pthread_mutex_lock(&shared_resource.lock);
```

```
    shared_resource.data = calloc(100);
```

```
    memset(shared_resource.data, 'A', 99);
```

```
    pthread_mutex_unlock(&shared_resource.lock);
```

```
    pthread_exit(NULL);
```

```
}
```

```
void *thread_two(void *a)
```

```
{
```

```
    pthread_mutex_lock(&shared_resource.lock);
```

```
    // use shared_resource.data
```

```
    pthread_mutex_unlock(&shared_resource.lock);
```

```
    pthread_exit(NULL);
```

```
}
```

```
struct RESOURCE_T{  
    char *data;  
    pthread_mutex_t lock;  
} shared_resource;
```



Return expressions

- ▶ The C99 standard states ““A return statement without an expression shall only appear in a function whose return type is void.”
- ▶ Sometimes functions that are not declared void return without an expression, this results in returning an uninitialized return value (whatever is in the EAX register).

Return expressions: *example*

```
int random_call(void)
{
    return 0xcafebabe;
}
int uninitialized_return_function(void)
{
    random_call();
    return;
}
int main(void)
{
    printf("0x%08x\n",
uninitialized_return_function());
    return;
}
```



Return expressions: *example*

```
int random_call(void)
{
    return 0xcafebabe;
}

int uninitialized_return_function(void)
{
    random_call();
    return 0;
}

int main(void)
{
    printf("0x%08x\n",
uninitialized_return_function());
    return;
}
```



-
- ▶ Any variable that is not initialized at declaration is considered “uninitialized” and worth auditing.
 - ▶ Most commonly, structures and arrays are where you’ll find these vulnerabilities, so look for any which haven’t first called `memset()`.
 - ▶ Audit conditional code paths which are used to initialize variables, and see what happens if they fail.
 - ▶ Audit any functions which initialize arguments, and any calls that don’t check a return value.
 - ▶ Keep track of any variable or memory that is used by multiple threads or signal handlers.





Exploiting

What are the challenges faced

- ▶ With the release of new Operating Systems, security technologies enabled by default are making exploitation of memory corruption vulnerabilities harder.
- ▶ For example, exploits must overcome:
 - ▶ Address Space Layout Randomization (ASLR)
 - ▶ SafeSEH (Structured Exception Handling)
 - ▶ Stack Smashing Protection (SSP)
 - ▶ No execute (NX)



What can be done?

- ▶ Uninitialized variables can help with bypassing some of these protections.
- ▶ Information leaks will assist with bypassing ASLR.
- ▶ Uninitialized objects or function pointers will assist with bypassing SSP.
- ▶ In some cases, applications can only be crashed.



Stack Delta/Reachability graphing

- ▶ Introduced by Halvar Flake at CanSecWest 2006.
- ▶ Provides a means of determining code paths which write to certain stack deltas.
- ▶ This is useful in determining the exploitability of local automatic variables.



Heap Feng Shui

- ▶ Introduced by Alexander Sotirov in 2007.
- ▶ Provides a means of controlling the state of the heap through Javascript.
- ▶ This is useful in exploiting browser based heap vulnerabilities.





AUTOMATING

Tools for automatically discovering

- ▶ The manual processes of keeping track of variable use can become tedious.
- ▶ There exists several compile time options that can help, and warn on any cases that are vulnerable.
- ▶ Not all vulnerabilities will be detected at compile time, so combining tools for maximum results is important.



Compile time: GCC

- ▶ **GCC:**

- ▶ `-Wuninitialized/-Winit-self`

- ▶ **Caveats:**

- ▶ Requires optimization flags to work.
- ▶ Not for volatile storage.
- ▶ Function calls with no return checking aren't found .
- ▶ Can't find use after free related issues.



Compile time: MSVC

▶ Compiler options:

- ▶ /analyze: Enable code analysis
- ▶ Compiler annotations to ensure function return values are checked.

```
[returnvalue:Post( MustCheck=SA_Yes )] double* CalcSquareRoot  
  
(  
  
    [Pre( Null=SA_No )] double* source,  
  
    unsigned int size  
  
);
```



Compile time: Phoenix Framework

- ▶ **Compiler options:**
 - ▶ -d2UninitializedLocal.dll
 - ▶ -d2WarnMayUninit
- ▶ There was a good example from EuSecWest earlier this year on detecting MS06-013 with this framework.



Dynamic analysis: MSVC

- ▶ **Compiler options:**
 - ▶ `/RTCu`: uninitialized local usage checks.
 - ▶ `/DEBUG /MDd`: link with the `MSVCRTD.lib` debug library.
- ▶ Provides debug versions of the heap allocator, and initializes all variables and memory with magic values.
- ▶ Any uninitialized variable/memory use raises an exception and can be debugged.



Dynamic analysis: Valgrind

- ▶ The address space is “mirrored”, and all access to uninitialized variables can be found with bit-precision.
- ▶ Unfortunately floating point, MMX, and SSE registers are not shadowed.



Conclusion

Conclusion

- ▶ Uninitialized variables can lead to exploitable vulnerabilities, and have been the cause of several well published security advisories.
- ▶ This bug class can be mostly be identified via the use of compile time warnings.
- ▶ In other cases, running your code through a debug runtime analysis tool will almost surely find uninitialized variable access.



Thankyou

▶ mercy@felinemenace.org

