Lecture Notes for seminar course (second half)

2012 Fall,

Ruizhong Wei

Review of Cryptography

- What is a block cipher and what are main techniques of modern block ciphers?
- Why some mode such as CBC, CFB are used for block cipher? Why use master key and session keys?
- What is a public key system? Why we need a public key system?
- What is a digital signature? What is message authentication?
- What is a secure hash function and how to use the hash function?
- Why random number generator is important in cryptography?
- What is PKI? What is a certificate?
- What is a password? How does the password work?

Part I: Operating System Security

Operation systems

Provides the interface between the users (applications) and the computer hardware (CPU, main memory, disk drives, input/output devices, network interfaces, etc.)

- Multi-users (with different levels) and multi-tasking (different priorities)
- The layers of a computer system: Hardware; The kernel and non-essential OS application (OS); And userland applications (use applications)
- Input/output devices drivers (keyboard, network card, USB port, etc).
- System call (contained in library). Allows applications to use predefined API's to communicate with kernel.

Processes (running programs in RAM)

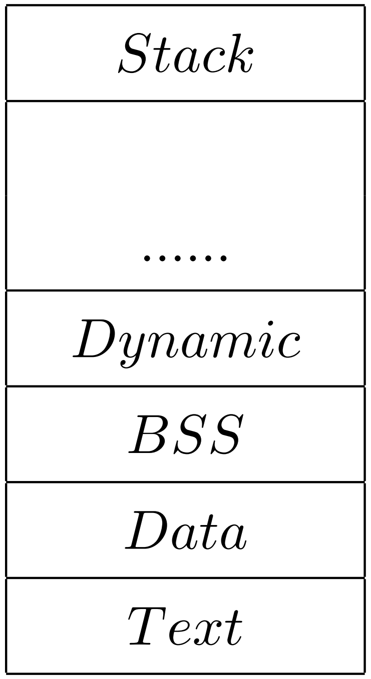
- time slicing capability for multiple processes.
- process tree (process forking: new processes are created by a process)
- process ID (PID), user ID (uid) and group ID (gid)
- inter-process communication (reading and writing files, share RAM), pipes and sockets
- signals (UNIX) and remote procedure calls (windows), interrupts.
- daemons (UNIX) and services (windows), (backgroud processes)

Filesystem

- file access control
- file permissions and permission matrix
- Unix file permissions

Memory management

- Address space (allocated a region of memory for a process executes), contiguous address space for each process.
- Unix memory model: Text (machine code of the program), Data (initialized static program variables), BBS (block started by symbol contains uninitialized static variables), Heap (dynamic segment stores data generated during a process), Stack (keeps tack of the call structure of subroutines and their arguments).


Stack
......
Dynamic
BSS
Data
Text

- Memory access permissions: Text usually is read-only, address is divided into user space and kernel space, a process usually is not allowed to access the address spaces of others.
- Virtual memory: use memory management unit to arrange memory so that processes are allowed to act as if their memory is contiguous (actually not). Sometimes external drive can be used to store data not using.
- Page faults: when a block of the address space is no accessed for an extended period of time, it may be paged out and written to disk. When a page fault occurs, paging supervisor reads the block back into RAM.

Virtual machine: host OS, guest OS and hypervisor (or virtual machine monitor).

- emulation: the host OS simulates virtual interfaces that the guest OS interacts with.
- virtualization: the virtual interfaces within the VM are matched with the actual hardware on the host machine.
- cloud computing

Process security

Monitor and protect the processes that are running on the computer.

Inductive Trust:

- The boot sequence: BIOS(basic input/output system), second-stage boot loader, rest of the OS.
- Use BIOS password to prevent a second-stage boot loader to be executed without proper authentication (boot by a live-CD).
- Boot device hierarchy determines the order of precedence of booting devices. Utilize second-stage boot loaders that require password protections that only allow authorized users to boot from external storage media.
- Hibernation file: used to stores entire contents of the machine's memory on disk so that the state of the computer can be quickly restore when the system is powered back on (`hiberfil.sys`). This file may be attacked by recovering the RAM content or modifying the hibernation file. Encryption should be used to protect hibernation files and swap files.

Monitoring:

- Event logging: `Windows Event Log (Isass.exe)` , Unix `/var/log` using `syslog` daemon.
- Processing monitoring: task manager in windows and `ps, top, pstree, kill` in Linux.
- Process explorer: task manager

Memory and file system security

Virtual memory security:

- Swap files: `pagefile.sys` for windows of `swap partition` for Linux, used for page file. Usually prevent users from viewing the contents of virtual memory files.
- Attacks on virtual memory: suddenly power off the machine and boot to another OS via external media may be possible to view the swap files and expose sensitive information. Hard disk encryption is used to prevent such an attack.

Password:

- Hash function and salt: password file stores $(U, S, h(S||P))$ where U is userid, S is salt (random string), P is password. Dictionary attack.
- Password authentication

  Windows: `NTLM` algorithm which is challenge-response protocol. Password are hashed and then used as key to encrypt (using DES) the random challenge value. File: `security accounts manager (SAM)`

  Unix: Using salt in password and MD5 or DES. Files: `/etc/passwd /etc/shadow`.
- Crack or John or other password crackers.
- One-time password

Access control

Access control decides what folders and files a user can access.

- Access control entries (ACE) and lists (ACL): ACE is a triplet `(principal, type, permission)`, where principal is either a user or a group, type is either allow or deny, permission is read, write, execution, etc. ACL is an ordered list of ACEs for each object. Disadvantage is they do not provide an efficient way to enumerate all the access rights for a given subject (e.g. when a user is removed)
- Capabilities: For each subject (user or group), list the objects for which it had nonempty access control rights, together with the specific rights. Disadvantage is the subject is not associated directly with objects.
- Linux permissions: Use file permission matrix. Path based access control principal. Owners of files can change the

permissions on those file (discretionary access control DAC). Append-only or immutable attributes. `chmod` command. Security-Enhanced Linux (SELinux) developed by NSA tried to use the principal of least privilege. In SELinux, users are not given the power to decide security attributes of their own files. They are decided by a central security policy administrator.

- Windows permissions: Use ACL model. Use inherited ACEs that may cause problem.
- How to set permission policies.
- The SetUID bit: used in the file permission matrix. If this bit is set, then that program runs with the effective user ID of its owner, rather than the process that executed it. **`setuid`** **`seteuid`** are used in Linux. This may cause security problem if the setuid program are not using safe programming practices.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>

static uid_t euid, uid;

int main(int argc, char* argv[])
{
 FILE *file;
 uid = getuid();
 euid = geteuid();
 seteuid(uid);
 ... ...
 seteuid(euid);  /*raise privileges*/
 file = fopen("home/admin/log", "a");
 seteuid(uid); /*drop privilege again*/
 fprintf(file, "Someone used this program. \n")’
 fclose(file);
 return 0;
}
```

Access Matrix:

	File 1	File 2	Process 1	Process 2
Process 1	read	read, write	read	–
Process 2	read, write	–	read,write	read

One disadvantage of access matrix is it will be a sparse data structure.

This access matrix model presents a problem for secure systems: untrusted processes can tamper with the protection system. Using protection state operations, untrusted user processes can modify the access matrix by adding new subjects, objects, or operations assigned to cells.

A protection system that permits untrusted processes to modify the protection state is called a discretionary access control (DAC) system. This is because the protection state is at the discretion of the users and any untrusted processes that they may execute.

A mandatory protection system is a protection system that can only be modified by trusted administrators via trusted software, consisting of the following state representations:

- A mandatory protection state is a protection state where subjects and objects are represented by labels where the state describes the operations that subject labels may take upon object labels;
- A labeling state for mapping processes and system resource objects to labels;
- A transition state that describes the legal ways that processes and system resource objects may be relabeled.

A label is simply an abstract identifierthe assignment of permissions to a label defines its security semantics. Labels are tamperproof because: (1) the set of labels is defined by trusted administrators using trusted software and (2) the set of labels is immutable. Trusted administrators define the access matrix's labels and set the operations that subjects of particular labels can perform on objects of particular labels. Such protection systems are mandatory access control (MAC) systems because the protection system is immutable to untrusted processes. Since the set of labels cannot be changed by the execution of user processes, we can prove the security goals enforced by the access matrix and rely on these goals being enforced throughout the system's execution.

Reference monitor

It takes a request as input, and returns a binary response indicating whether the request is authorized by the reference monitor's access control policy. Three distinct components of a reference monitor:

- its interface(where the system queries made to the reference monitor)
- its authorization module (takes interface's inputs (e.g., process identity, object references, and system call name), and converts these to a query for the reference monitor's policy store.)
- its policy store (is a database for the protection state, labeling state, and transition state)

The reference monitor concept defines a system:

- Complete mediation: its access enforcement mechanism mediates all security-sensitive operations.
- Tamperproof: its access enforcement mechanism, including its protection system, cannot be modified by untrusted processes.
- Verifiable:The access enforcement mechanism, including its protection system, must be small enough to be subject to analysis and tests, the completeness of which can be assured. That is, we must be able to prove that the system enforces its security goals correctly

Role-based access control (RBAC)

Access control based on roles instead of individual users, which can simplify the access control in many cases. However, most OS does not implement this.

Core RBAC: users are assigned to roles, permissions are assigned to roles and users acquire permissions by being members of roles. Many-to-many assignment. User-role and permission-role reviews. User sessions.

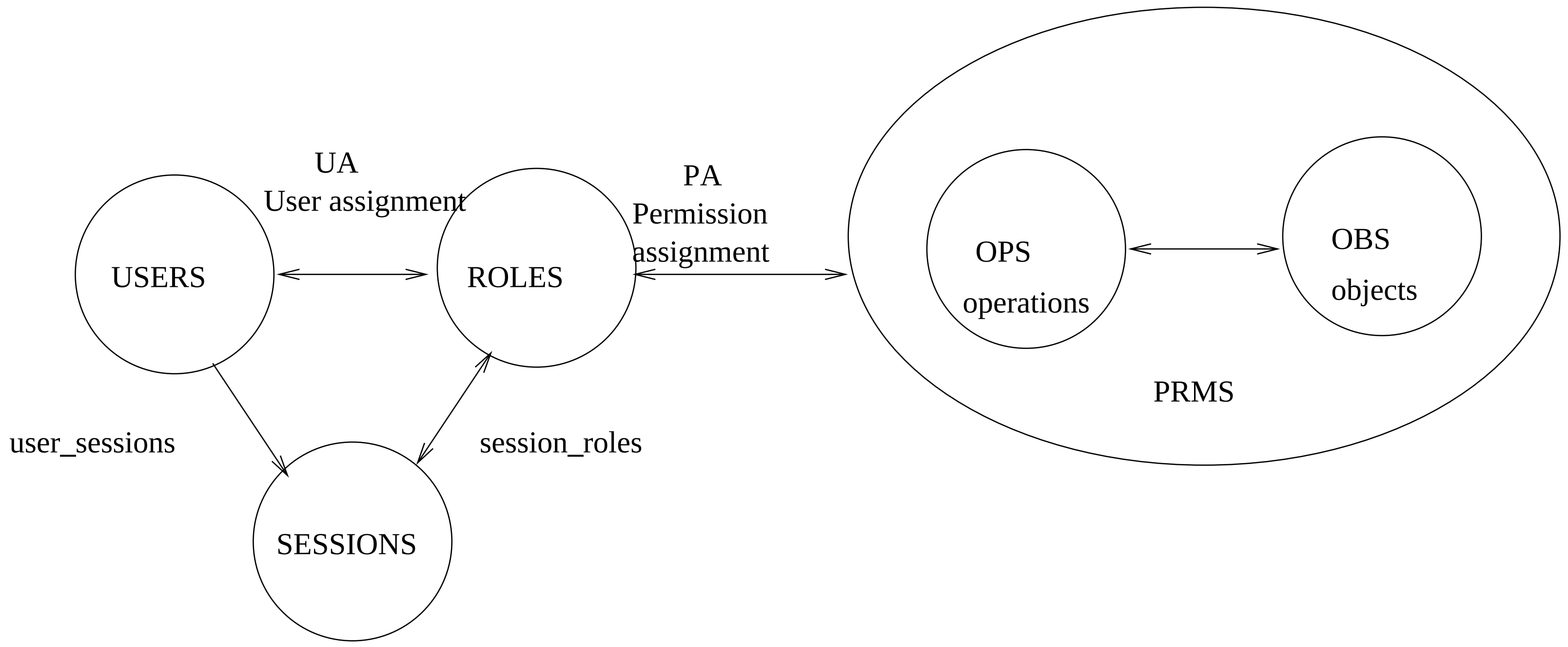


Figure 1: The core RBAC

Hierarchical RBAC: roles are arranged in hierarchies. Senior roles acquire the permission of their juniors and junior roles acquire the user membership of their seniors.

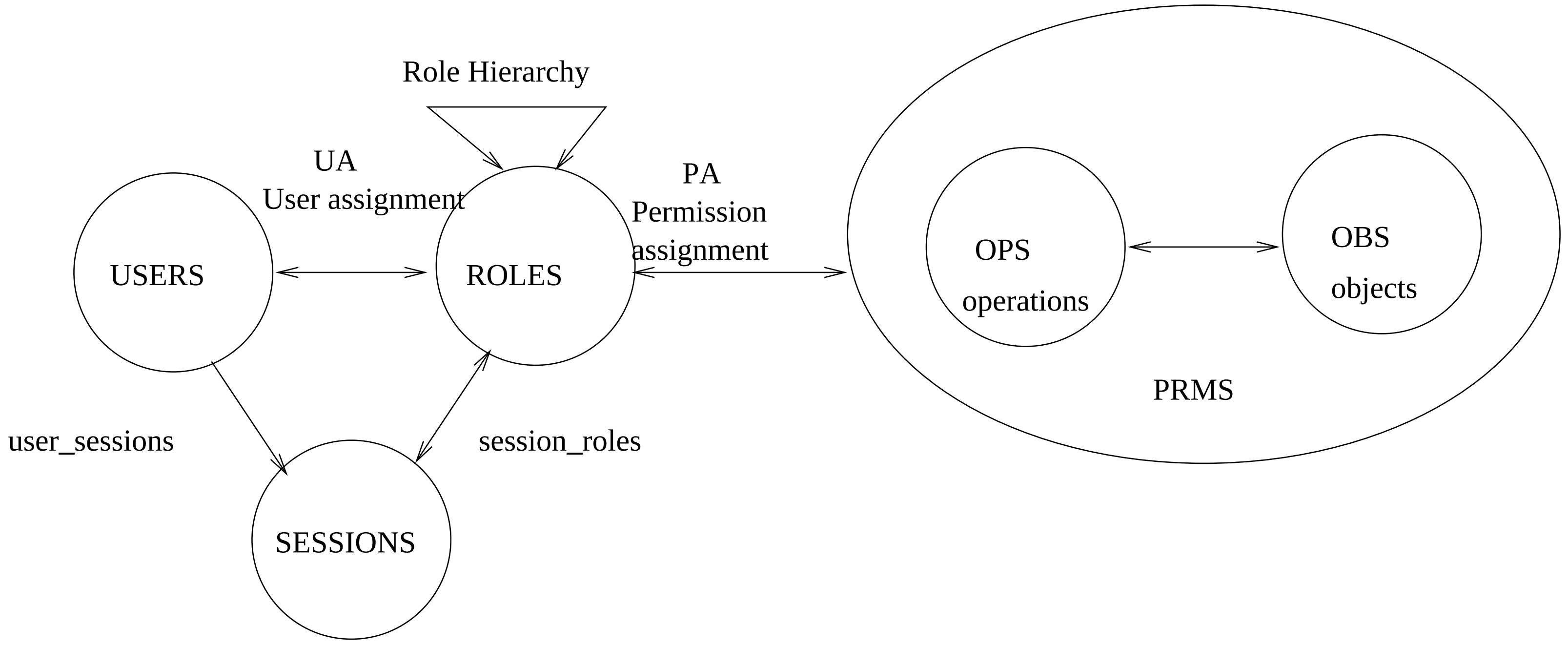


Figure 2: The Hierarchical RBAC

- Static separation of duty (SSD): place constraints on the assignment of users to roles. Membership in one role may prevent the use from being a member of one or more other roles. SSD in the presence of a hierarchy is also considered.
- Dynamic separation of duty (DSD): limits the availability of the permissions by placing constraints on the roles that can be activated within or across a user's sessions.

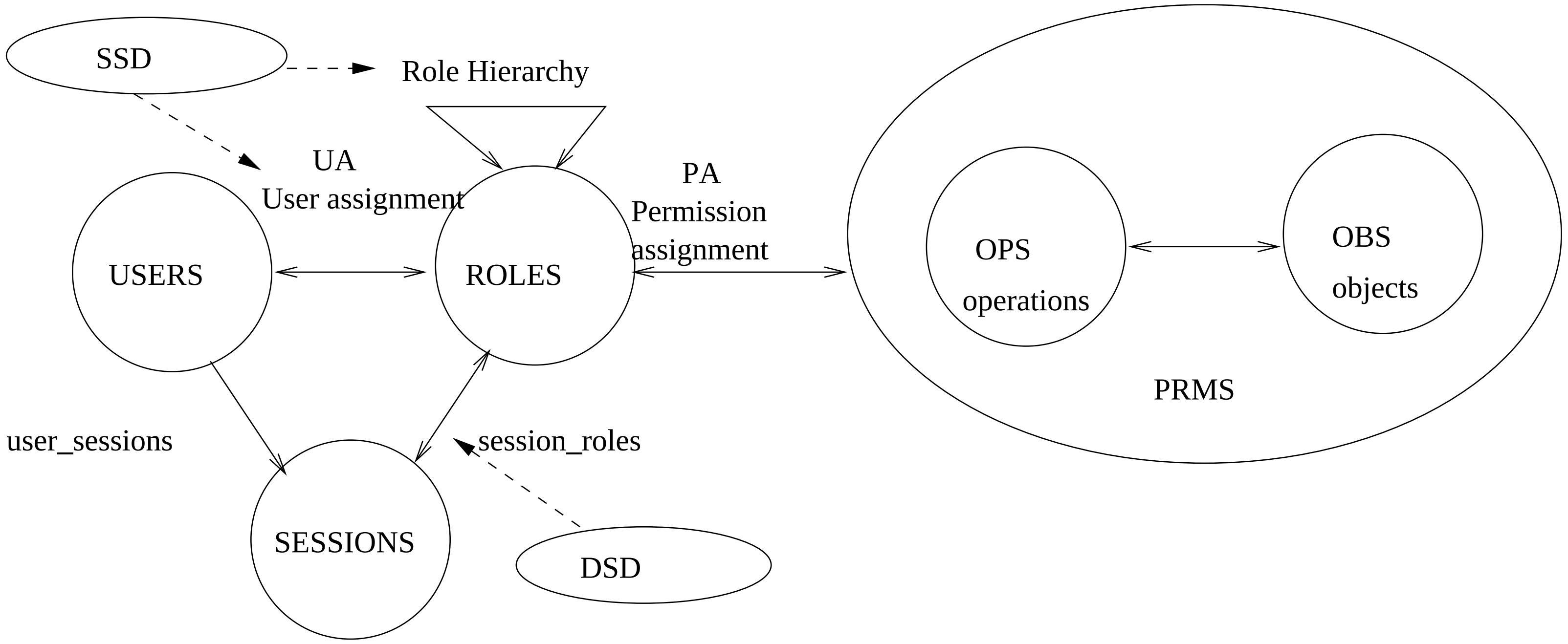


Figure 3: The Constrained RBAC

File descriptors

- When a program needs to access a file, a call is made to the `open` system call, which results in the kernel creating a new entry in the `file descriptor table` which maps to the file's location on the disk. When receiving a read or write system call, the kernel looks up the file descriptor in the table and perform the read or write at the appropriate location on disk.
- When a process creates a child process, that child process inherits copies of all of the file descriptors that are open in the parent. This may cause a security problem known as a file descriptor leak.

Example

```
#include <stdio.h>
#include <unistd.h>

int main( int argc, char*argv[])
{
 FILE *passwords;
 passwords = fopen("/home/admin/passwords", "r");
 .......
 .......
 execl("/home/joe/shell", "shell", NULL);
 ......
}
```

To fix the vulnerability, a call to `fclose` should be made before executing the new program.

Symbolic links

Symbolic links (symlink) can be a chain. An attacker could trick a program by creating a symlink to a secure file and specifying the path of the symlink instead. When open files, use a `stat` system call to retrieves information on files.

Windows uses shortcut.

Virtual machine system security: Some problems need to consider:

- virtual machine systems may generate problems for administrators, as there will be many more VMs to administer and disinfect than physical machines.
- the ability to save, restore, and migrate VMs may generate difficulties in ensuring that the VM software base is current and consistent with organizational requirements.
- the identity of virtual machines will be harder to determine than the identity of physical machines. Virtual machines may even be able to migrate across administrative domains.
- data leaks of VMs into various physical systems and the integrity impact of individual systems into VMs may be difficult to track.

Application program security

Compiling and linking:

- Statically linking: safer but require more space and difficult debugging
- Dynamically linking: when the program is executed, the loader determines which shared libraries are needed for the program, finds them on disk, and imports them into the process's address space.
- DLL injection: injecting codes to DLL (dynamic linking library) so that no recompiling is needed. May cause security problem.

Simple buffer overflow attacks: When a program allocates a fixed-size buffer in memory in which to store information, it is important that copying user-supplied data to this buffer is done securely and with boundary check.

Arithmetic overflow: caused by the representation of integers in memory. Add a large positive integer might result a negative.

```
#include<stdio.h>

int main(int argc, char * argv[])
{
 unsigned int connections = 0;
 ... ...
 if(connections < 5)  //missing this will cause problem
 connections++;
 if(connections < 5)
   grant_access();
 else
  deny_access();
 return1;
}
```

Making huge number of connections will cause counter overflows and wraps around to zero.

Stack-based buffer overflow

A stack is a contiguous block of memory containing data. A register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time.

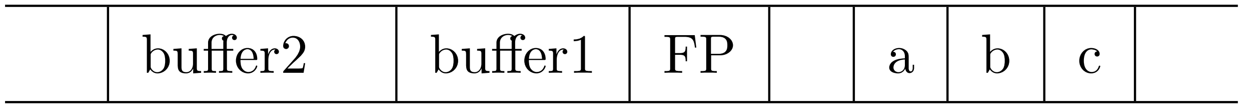
The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.

A frame pointer (FP) points to a fixed location within a frame. Many compilers use FP for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs.

The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit). Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables.

Example 1

```
void function (int a, int b, int c)
{
   char buffer1[5];
   char buffer2[10];
}
int main()
{
  function(1,2,3);
}
```



Example

```
void function(int a, int b, int c){
  char buffer1[5];
  char buffer2[10];
  int *ret;
  ret=buffer1+12;
  (*ret)+=8;
}

void main(){
 int x;
 x=0;
 function(1,2,3);
 x=1;
 printf("%d\n",x);
}
```

What we have done is add 12 to buffer1[]'s address. This new address is where the return address is stored. We want to skip pass the assignment to the printf call. How did we know to add 8 to the return address? We used a test value first (for example 1), compiled the program, and then started gdb:

```
[aleph1]$ gdb example3 GDB is free software and you are welcome to
distribute copies of it under certain conditions; type "show
copying" to see the conditions. There is absolutely no warranty for
GDB; type "show warranty" for details. GDB 4.15
(i586-unknown-linux), Copyright 1995 Free Software Foundation,
Inc... (no debugging symbols found)...

(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
```

```
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
```

```
0x80004c7 <main+55>:    nop
```

We can see that when calling function() the RET will be 0x8004a8, and we want to jump past the assignment at 0x80004ab. The next instruction we want to execute is the at 0x8004b2. A little math tells us the distance is 8 bytes.

Some techniques used by the attackers.

- NOP sledding: To get around needing the actual address, the malicious command is often padded on both sides by NOP (no-op) instructions, a type of pointer. Padding on both sides is a technique used when the exact memory range is unknown. Therefore, if the address the hacker specifies falls anywhere within the padding, the malicious command will be executed.
- Trampolining: Try to use common external libraries which usually loaded into predictable memory locations. For example, from some assembly code instruction in a DLL, the attacker might find instruction tells the processor to jump to the address stored in somewhere.

- The return-to-libc attack: the attacker found the address of a C library function, such as `system()` or `execv`. The attacker then use buffer overflow to overwriting the return address with the address of the library function. Following this address, the attacker provide a new address that the libc function will return to when it is finished.

Heap-based buffer overflow

Memory leak problems: if programers allocate memory on the heap and do not explicitly de-allocate that block, it remains allocated but are not actually being used, which allowing an attacker to execute arbitrary code by buffer overflow.

Heap-based overflows are generally more complex than stack-based buffer overflows and require a more in-depth understanding of how garbage collection and heap are implemented.

```
#include <stdio.h>
#include <stdlib.h>
#include <sting.h>

int main(int argc, char *argv[])
{
 char *buf = malloc(256);
 char *buf2 = malloc(16);
 strcpy(buf, arbv[1]); //this may cause heap buffer overflow.
 printf("argument: %s\n", buf);
 ......
}
```

Prevent buffer overflow attacks

Abuffer overflow attack requires two things. First, a buffer overflow must occur in the program. Second, the attacker must be able to use the buffer overflow to overwrite a security sensitive piece of data (a security flag, function pointer, return address, etc).

If we want to prevent buffer overflows completely we must stop one of these two things, i. e. either:

1. Prevent all buffer overflows or
2. Prevent all sensitive information from being overwritten

Both these solutions are costly in terms of efficiency and many programs therefore settle for a partial goal, such as:

- Prevent use of dangerous functions: gets(), strcpy(), etc.
- Prevent return addresses from being overwritten
- Prevent data supplied by the attacker from being executed (stops the attacker from jumping into his own buffer)

Language Tools

One solution at the language level is to switch to a language that provides automatic bounds checking of buffers, such as Java, Perl or Python. However, in most projects this is not an option.

A better solution is to use a library module that implements "safe", bounds-checked buffers, such as the standard C++ string module or `libmib` (Software Component Library).

- it requires a complete rewrite of all the source code for the project.
- most programs have to interface with prewritten library code.

Some programs have to convert between "safe" and "unsafe" buffers. Whenever a buffer is in the "unsafe" mode, buffer overflow problems can occur. There is also a risk that programmer's will forget to convert buffers back to the "safe" mode.

An alternative to making every buffer access safe is to target only those specific functions in C which are known to be dangerous, e. g. strcpy(), strcat(), gets() and sprintf().

The simplest solution to securing the dangerous function is to disallow them. They all have ”safe” counterparts that can be substituted, such as strncpy() and snprintf(), which in addition to the buffers also take a size parameter.

To make sure that the unsafe versions are not used by mistake, their prototypes can be removed from the header files.

The `libsafe` library from Bell Labs provides a way of securing calls to these functions, even if the source code is not available. It does this by replacing the implementation of the dangerous functions in the shared libc library with safe versions.

`libsafe` makes use of the fact that stack frames are linked together by frame pointers (this is implementation dependent, but many C compilers on many platforms use this solution). When a buffer is passed as argument to one of the unsafe functions, libsafe follows the frame pointers to find the stack frame where the buffer was allocated (if it is not found, it is assumed that the buffer resides on the heap). It then checks the distance to the closest return address on the stack. When the function executes it makes sure that this address is not overwritten. If an attempt to overwrite this address is made, the program terminates with a vulnerability warning.

Source Code Tools

A source code tool analyzes the source code of a program and tries to determine whether it contains any dangerous constructions that could lead to buffer overflows.

We cannot expect a source code tool to be able to detect all possible instances of buffer overflow while at the same time yielding no false positives, since that is a task of the same difficulty as solving the halting problem. Constructing good source code tools will therefore always be a heuristic task.

The source code tools available today make only a limited, local analysis of the code and are therefore heavily restricted in the types of buffer overflow problems that they can detect.

ITS4 (Software Security Tool) is a source code tool that checks for the use of dangerous function. It is a bit smarter than a plain grep search in that it can rule out some cases where the use of a dangerous function usually does not pose a problem.

An analyzer that could perform more extensive, cross-function analysis would be a valuable tool in a security audit. Constructing such a tool would however be a major undertaking. The tool would have to try to keep track of the size of all buffers and the possible ranges of variables to be able to detect when a buffer overflow might occur.

Dynamic analysis tools such as `Purify` are an alternative to static analysis tools. A dynamic analysis tool analyzes the memory use of a program as it is run. Dynamic analysis tools can detect buffer overflow problems if they occur in a test run of a program. However, errors that occur in test runs can usually be detected even without an analysis tools, since they typically cause the program to crash. The main advantage of dynamic analyzers is that they allow for the error to be swiftly located once it has been detected.

Compiler Tools

Buffer overflows can be prevented by adding bounds checking to all buffers. To do this, the compiler must add code for keeping track of the size of buffers and for checking that every buffer access falls within the allocated size. Herman ten Brugge has written a patch that adds bounds checking to the gcc compiler. Another project for adding bounds checking to gcc is managed by Greg McGary.

The problem with adding bounds checking to every pointer is that it results in a great performance hit. Code size and execution time may grow by 200 or more.

Instead of preventing all buffer overflows we might try to protect the return address from being overwritten.

One possible way of doing this is to write the return address to a "safe" place (far from the local buffers) at the start of a function and then restore it just before the function returns. Since the function can call other functions which in turn need to store their return addresses we will need a stack to keep track of all the stored return addresses. In practice this solution thus means separating the stack used for return addresses from the stack used for local variables. This requires a lot of changes to the compiler. A program called StackShield implements this approach, which is also a GNU C compiler extension.

Instead of moving the return address, we can move the buffers. For example, we could allocate all buffers in heap space instead of in stack space. The disadvantage of this solution is that the heap is substantially slower than the stack.

A slightly different approach is taken by the StackGuard program. StackGuard does not prevent the return address from being overwritten, instead it tries to detect when it happens and take the appropriate action (terminating the program before any damage is done).

StackGuard accomplishes this in an ingenious way. Whenever a function is called, code is added for pushing a small value, called a “canary” value, to the stack. This value thus ends up between the local variables and the return address.

	buffer2	buffer1	FP	ca	ret	a	b	c	

When the function exits it checks that the canary value has not been modified before returning. The idea is that a buffer overflow in one of the local variables cannot overwrite the return address without simultaneously destroying the integrity of the canary value. It thus becomes possible to detect whether a buffer overflow has occurred before the function returns.

For this to work, the attacker must not be able to guess the canary value. If the attacker can correctly guess the canary value, he can overwrite the return address without being detected. StackGuard can set the canary value in one of two possible ways. A random value may be used, which is hard for the attacker to guess or the value 0 may be used, which is easy to guess but hard for the attacker to put into his buffer (since most sensitive buffer operations, such as string copying, are terminated by a zero value).

StackGuard can only protect against stack attacks, not against variable attacks, but there is ongoing work to add canary values to function pointers too, since they are also a vulnerable target. The added canary checks increase function call and return times with 40 - 80 %. If the program contains many small function calls and inlining is not used, the total overhead can be in this range. If the program does not contain as many function calls, the overhead will be smaller.

IBM's stack-smashing protector (ssp), originally named ProPolice, is a variation of StackGuard's approach. Like StackGuard, ssp uses a modified compiler (gcc) to insert a canary in function calls to detect stack overflows. However, it adds some interesting twists to the basic idea. It reorders where local variables are stored, and copies pointers in function arguments, so that they're also before any arrays. This strengthens the protection of ssp; this means a buffer overflow can't modify a pointer value (otherwise an attacker who can control a pointer can control where the program saves data using the pointer). By default, it doesn't instrument all functions, only those that it deems as being in need of protection (mainly functions with character arrays).

In theory, this could weaken the protection slightly, but this default improves performance while still protecting against most problems. As a practical matter, they implemented their approach using gcc in a way that is architecture-independent, making it easier to deploy. The widely-respected OpenBSD, which concentrates on security, uses ssp (also known as ProPolice) across their entire distribution as of their May 2003 release.

Microsoft has added a compiler flag (/GS) to implement canaries in its C compiler, based on the StackGuard work.

Operating System Tools

Solar Designer has created a patch for Linux that makes the stack non-executable. This means that the attacker can no longer inject his own code into the stack and run it. The patch also maps libc to the 0x00... memory range, so that it is impossible to call libc functions without having a zero in the buffer.

It is known that attackers often can't insert the ASCII NUL character (0) using typical buffer overflow attacks. That means that attackers find it difficult to make a program return to an address with a zero in it. Since that's the case, moving all executable code to addresses with a 0 in it makes attacking the program far more difficult.

The largest contiguous memory range with this property is the set of memory addresses from 0 through 0x01010100, so that's been christened the "executable region" (there are other addresses with this property, but they're scattered). Combined with non-executable stacks, this is pretty valuable: non-executable stacks prevent attackers from sending new executable code, and "executable region" makes it hard for attackers to work around it by exploiting existing code. This protects against stack, buffer, and function pointer overflows, all without recompilation.

However, this method doesn't work for all programs; big programs may not fit in that region (so the protection will be imperfect), and sometimes attackers can get a 0 into their destination.

The disadvantage of having a non-executable stack is that some legitimate programs (though not very many) actually execute content on the stack. These programs will cease to work if the patch is applied.

Another approach is called “split control and data stack” – the idea is to split the stack into two stacks, one to store control information (such as the “return” address) and the other for all the other data. Xu et al. implement this in gcc, and StackShield implements it in the assembler. This makes it much harder to manipulate the return address, but it doesn’t defend against buffer overflow attacks that change the data of calling functions.

In fact, there are other approaches as well, including randomizing the locations of executables; Crispen’s “PointGuard” extends the canary idea to the heap, and so on. Figuring out how to defend today’s computers has become an active research task.

Buffers have a limited amount of space. So there are really two major possibilities for dealing with running out of space.

"Statically allocated buffer" approach: When the buffer runs out, that's it; you complain and refuse to add anything more to the buffer. "Dynamically allocated buffer" approach: When the buffer runs out, you dynamically resize the buffer to a larger size until you run out of total memory.

There are disadvantages to the static approach. In fact, the static approach may sometimes create a different vulnerability! Static approaches basically throw away “excess” data. If the program uses the resulting data anyway, an attacker will try to fill up the buffer so that when the data is truncated, the attacker will fill the buffer with what the attacker wanted. If you’re using a static approach, you should ensure that the worst an attacker could do won’t invalidate some assumption, and a few checks on the final result would be a good idea too.

Dynamic approaches have lots of advantages: they can scale up to larger problems (instead of creating arbitrary limits), and they don't have the problem of truncations causing security problems. But they have problems of their own. When arbitrarily sized data is accepted, you may run out of memory – and that may not happen just during input. Any memory allocation can fail, and it's not easy to write C or C++ programs that truly handle that well. Even before truly running out of memory, you can cause the computer to get so busy that it becomes useless. In short, dynamic approaches often make it much easier for attackers to create denial-of-service attacks. So you'll still need to limit inputs. What's more, you'll have to carefully design your program to handle memory exhaustion in arbitrary places, which is not easy.

Format string attacks

The `printf` family of C library functions can be used to write the numbers of bytes to memory. When a program does not supply a format string, but is user-supplied, then an attacker could carefully craft an input that uses format strings to write to arbitrary locations in memory (return address, function pointer, etc.).

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 printf("Your argument is : \n");
 printf(argv[1]); //should be: printf("%s", argv[1]);
}
```

Race conditions

A race condition is any situation where the behaviour of the program is unintentionally dependent on the timing of certain events.

```
#include <stdio.h>
......
int main(int argc, char *argv[])
{
 int file;
 char buf[1024];
 memset(buf, 0, 1024);
 if(argc < 2) {
  printf("Usage: printer [filename]\n");
  exit(-1);
  }
 if(access(argv[1],R_OK) !=0) {
  printf("Cannot access file.\n");
  exit(-1);
  }
 file = open(argv[1],O_RDONLY);
 read(file,buf, 1023);
   close(file);
 printf("%s\n",buf);
 return 0;
}
```

There is a tiny time delay between the calls to `access()` and `open`. An attacker could exploit this small delay by changing the file in question between the two calls. For example, suppose the attacker provided a innocent text file `/home/jeo/try.txt` as an argument. After the call to `access()` returns 0, the attacker can quickly replace `/home/jeo/try.txt` with a symbolic link to `/etc/passwd` (use a program running at background to switches the link between these files repeatedly.)

The call of `access()` should be avoided:

```
#include <stdio.h>
......
int main(int argc, char *argv[])
{
 int file;
 char buf[1024];
 uid_t uid, euid;
 memset(buf, 0, 1024);
 if(argc < 2) {
  printf("Usage: printer [filename]\n");
  exit(-1);
  }
  euid = geteuid();
  uid = getuid();
 seteuid(uid); //drop privileges
 file = open(argv[1],O_RDONLY);
 read(file,buf, 1023);
   close(file);
seteuid(euid);
 printf("%s\n",buf);
 return 0;
}
```