CS107, Lecture 7
C Strings, Buffer Overflows and Security

Reading: K&R (1.6, 5.5, Appendix B3) or Essential C section 3

CS107 Topic 2

How can a computer represent and manipulate more complex data like text?

Why is answering this question important?

- Shows us how strings are represented in C and other languages (last time)
- Helps us better understand buffer overflows, a common bug (this time)
- Introduces us to pointers, because strings can be pointers (next time)

assign2: implement 2 functions a 1 program using those functions to find the location of different built-in commands in the filesystem. You'll write functions to extract a list of possible locations and tokenize that list of locations.

Learning Goals

- Understand how to use the built-in string functions for common string tasks
- Learn more about the risks of buffer overflows and how to mitigate them

Lecture Plan

- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows and Security

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

Lecture Plan

- **Recap: Strings so far**
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows and Security

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

C Strings

C strings are arrays of characters ending with a **null-terminating character** '\0'.

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
value	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`','`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'!'`	`'\0'`

String operations such as `strlen` use the null-terminating character to find the end of the string.

Side note: use `strlen` to get the length of a string. Don’t use `sizeof`!

Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***`str1, str2`***`)`, `strncmp(`***`str1, str2, n`***`)`	compares two strings; returns 0 if identical, <0 if ***str1*** comes before ***str2*** in alphabet, >0 if ***str1*** comes after ***str2*** in alphabet. ***strncmp*** stops comparing after at most ***n*** characters.
`strchr(`***`str, ch`***`)` `strrchr(`***`str, ch`***`)`	character search: returns a pointer to the first occurrence of ***ch*** in ***str***, or ***NULL*** if ***ch*** was not found in ***str***. `strrchr` find the last occurrence.
`strstr(`***`haystack, needle`***`)`	string search: returns a pointer to the start of the first occurrence of ***needle*** in ***haystack***, or ***NULL*** if ***needle*** was not found in ***haystack***.
`strcpy(`***`dst, src`***`)`, `strncpy(`***`dst, src, n`***`)`	copies characters in ***src*** to ***dst***, including null-terminating character. Assumes enough space in ***dst***. Strings must not overlap. **`strncpy`** stops after at most ***n*** chars, and <u>does not</u> add null-terminating char.
`strcat(`***`dst, src`***`)`, `strncat(`***`dst, src, n`***`)`	concatenate ***src*** onto the end of ***dst***. **`strncat`** stops concatenating after at most ***n*** characters. <u>Always</u> adds a null-terminating character.
`strspn(`***`str, accept`***`)`, `strcspn(`***`str, reject`***`)`	**`strspn`** returns the length of the initial part of ***str*** which contains <u>only</u> characters in ***accept***. **`strcspn`** returns the length of the initial part of ***str*** which does <u>not</u> contain any characters in ***reject***.

Substrings

We can combine pointer arithmetic and copying to make any substrings we'd like.

```
// Want just "ace"
char str1[8];
strcpy(str1, "racecar");

char str2[4];
strncpy(str2, str1 + 1, 3);
str2[3] = '\0';
printf("%s\n", str1);         // racecar
printf("%s\n", str2);         // ace
```

char * vs. char[]

- char * is an 8-byte pointer – it stores an address of a character
- char[] is an array of characters – it stores the actual characters in a string
- When you pass a char[] as a parameter, it is automatically passed as a char * (pointer to its first character)

char * vs. char[]

`char myString[]`

vs

`char *myString`

You can create char * pointers to point to any character in an existing string and reassign them since they are just pointer variables. You **cannot** reassign an array.

```
char myString[6];
strcpy(myString, "Hello");
myString = "Another string";                    // not allowed!
---
char *myOtherString = myString;
myOtherString = somethingElse;                  // ok
```

Lecture Plan

- **Recap:** Strings so far
- **Searching in Strings**
- **Practice:** Password Verification
- Buffer Overflows and Security

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```
char bailey[7];
strcpy(bailey, "Bailey");
char *letterI = strchr(bailey, 'i');
printf("%s\n", bailey);          // Bailey
printf("%s\n", letterI);         // iley
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the *first* one. Use `str`**r**`chr` to obtain a pointer to the *last* occurrence.

Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```
char bailey[11];
strcpy(bailey, "Bailey Dog");
char *substr = strstr(bailey, "Dog");
printf("%s\n", bailey);            // Bailey Dog
printf("%s\n", substr);            // Dog
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char bailey[10];
strcpy(bailey, "Bailey Dog");
int spanLength = strspn(bailey, "aBeoi");      // 3
```

"How many places can we go in the first string before I encounter a character <u>not in</u> the second string?"

String Spans

`strcspn` (c = "complement") returns the *length* of the initial part of the first string which contains only characters not in the second string.

```
char bailey[10];
strcpy(bailey, "Bailey Dog");
int spanLength = strcspn(bailey, "driso");     // 2
```

"How many places can we go in the first string before I encounter a character in the second string?"

str[c]spn vs. strstr

strspn/strcspn can't search for substrings because it does not pay attention to the order of the characters in the second string. **strstr** allows us to search for substrings within another string.

```
// these are all equivalent
int spanLength = strcspn(bailey, "driso");
int spanLength = strcspn(bailey, "sirdo");
int spanLength = strcspn(bailey, "odris");
```

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char ***. We can still operate on the string the same way as with a char[].

```
int doSomething(char *str) {
    char secondChar = str[1];
    ...
}

// can also write this, but it is really a pointer
int doSomething(char str[]) { ...
```

Arrays of Strings

We can make an array of strings to group multiple strings together:

```
char *stringArray[5];   // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```
char *stringArray[] = {
     "Hello",
     "Hi",
     "Hey there"
};
```

Arrays of Strings

We can access each string using bracket syntax:

```
printf("%s\n", stringArray[0]);  // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the first element of the array*. This is what **argv** is in **main**! This means we write the parameter type as:

```
void myFunction(char **stringArray) {

// equivalent to this, but it is really a double pointer
void myFunction(char *stringArray[]) {
```

Practice: Password Verification

Write a function **`verifyPassword`** that accepts a candidate password and certain password criteria and returns whether the password is valid.

```
bool verifyPassword(char *password, char *validChars, char
*badSubstrings[], int numBadSubstrings);
```

password is valid if it contains only letters in **validChars**, and does not contain any substrings in **badSubstrings**.

Practice: Password Verification

```
bool verifyPassword(char *password, char *validChars, char
*badSubstrings[], int numBadSubstrings);
```

Example:

```
char *invalidSubstrings[] = { "1234" };

bool valid1 = verifyPassword("1572", "0123456789",
          invalidSubstrings, 1);      // true
bool valid2 = verifyPassword("141234", "0123456789",
          invalidSubstrings, 1);      // false
```

Practice: Password Verification

`verify_password.c`

Lecture Plan

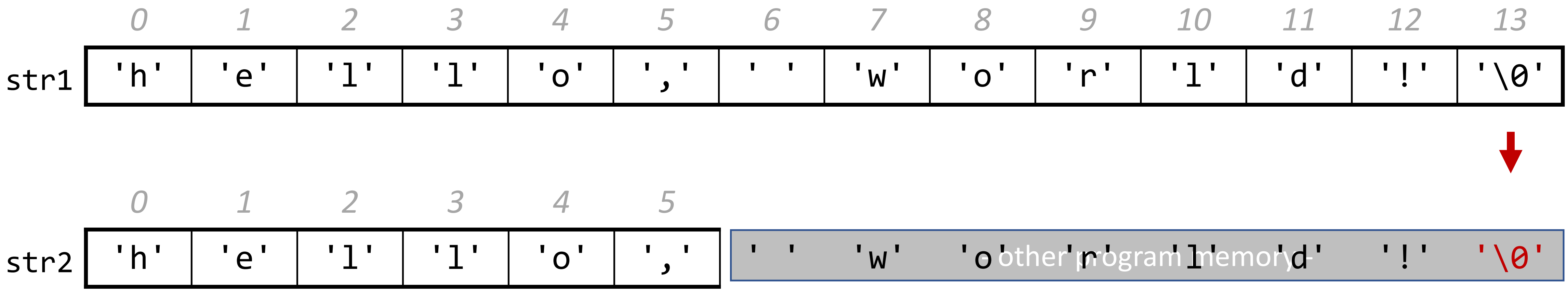
- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- **Buffer Overflows and Security**

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

Recall: Buffer Overflows

We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*. Writing past memory bounds is called a “buffer overflow”. It can allow for security vulnerabilities!

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```



Buffer Overflow Impacts

Buffer overflows are not merely functionality bugs; they can cause a range of unintended behavior:

- Let the user access memory they shouldn’t be able to access
- Let the user modify memory they shouldn’t be able to access
  - Change a value that is used later in the program
  - User changes the program to execute their own custom instructions instead
  - And more…

It’s our job as programmers to find and fix buffer overflows and other bugs not just for the functional correctness of our programs, but to protect people who use and interact with our code.

Buffer Overflow Example: ./buf

```
// first argument is the name, second is the password
int main(int argc, char *argv[]) {
    char secret[4] = "123";
    // assume secret comes right after name in memory
    // (this is not always true)
    char name[4];
    strcpy(name, argv[1]);

    if (!strcmp(secret, argv[2])) {
        printf("You're in!\n");
    }
    return 0;
}
```

name				*secret*			
?	?	?	?	'1'	'2'	'3'	'\0'

Which of these arguments would cause the program to print "You're in!"?

A. ./buf abcdefg efg
B. ./buf abcd abcd
C. ./buf a a
D. ./buf abcdefgh abcd

Respond on PollEv: pollev.com/cs107
or text CS107 to 22333 once to join.

Which of these arguments would cause the program to print "You're in!"?

./buf abcdefg efg — 0%

./buf abcd abcd — 0%

./buf a a — 0%

./buf abcdefg abcd — 0%

Buffer Overflow Example: ./buf

```
// first argument is the name, second is the password
int main(int argc, char *argv[]) {
    char secret[4] = "123";
    // assume secret comes right after name in memory
    // (this is not always true)
    char name[4];
    strcpy(name, argv[1]);

    if (!strcmp(secret, argv[2])) {
        printf("You're in!\n");
    }
    return 0;
}
```

name				secret			
?	?	?	?	'1'	'2'	'3'	'\0'

Which of these arguments would cause the program to print "You're in!"?

A. **./buf abcdefg efg**
B. ./buf abcd abcd
C. ./buf a a
D. ./buf abcdefgh abcd

Buffer Overflow Example: ./buf

```c
// first argument is the name, second is the password
int main(int argc, char *argv[]) {
    char secret[4] = "123";
    // assume secret comes right after name in memory
    // (this is not always true)
    char name[4];
    strcpy(name, argv[1]);

    if (!strcmp(secret, argv[2])) {
        printf("You're in!\n");
    }
    return 0;
}
```

name				*secret*			
'a'	'b'	'c'	'd'	'e'	'f'	'g'	'\0'

Which of these arguments would cause the program to print “You’re in!”?

A. **./buf abcdefg efg**
B. ./buf abcd abcd
C. ./buf a a
D. ./buf abcdefgh abcd

Buffer Overflow Impacts

- AOL instant messenger buffer overflow: allowed remote attackers to execute code: https://www.cvedetails.com/cve/CVE-2002-0362/
- Morris Worm: first internet worm to gain widespread attention; exploited buffer overflow in Unix command called ”finger”: https://www.zdnet.com/article/the-morris-worm-internet-malware-turns-25/

How can we fix buffer overflows?

There’s no single solution to fix all buffer overflows; instead, it’s a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How can we fix buffer overflows?

There's no single solution to fix all buffer overflows; instead, it's a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- <u>**Carefully reading documentation**</u>
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How Can We Fix Overflows?

Documentation & MAN Pages (Written by Others)

> “The strcpy() function copies the string pointed to by src, including the terminating null byte (‘\0’), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. Beware of buffer overruns! (See BUGS.) …
>
> BUGS
>
> If the destination string of a strcpy() is not large enough, then anything might happen. **Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine.** Any time a program reads or copies data into a buffer, the program first needs to check that there’s enough space. **This may be unnecessary if you can show that overflow is impossible, but be careful: programs can get changed over time, in ways that may make the impossible possible.**”

How can we fix buffer overflows?

There's no single solution to fix all buffer overflows; instead, it's a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- **<u>Thorough testing to uncover issues before release</u>**
- Thorough documentation to document assumptions in your code
- (Where possible) use of tools that reduce the possibility for buffer overflows

How Can We Fix Overflows?

- **Valgrind**: Your Greatest Ally
- Write your own tests
- Consider writing tests *before* writing the main program

✨ `cs107.stanford.edu/testing.html` ✨

How can we fix buffer overflows?

There's no single solution to fix all buffer overflows; instead, it's a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- **Thorough documentation to document assumptions in your code**
- (Where possible) use of tools that reduce the possibility for buffer overflows

Code Documentation

```
/* Function: myFunction
 * This function assumes that the provided string is
 * at most length 10.
 * ...
 */
void myFunction(char *str) {
    ...
```

How can we fix buffer overflows?

There's no single solution to fix all buffer overflows; instead, it's a combination of techniques to avoid them as much as possible:

- Constant vigilance while programming (checking arrays and where they are modified)
- Carefully reading documentation
- Thorough testing to uncover issues before release
- Thorough documentation to document assumptions in your code
- **<u>(Where possible) use of tools that reduce the possibility for buffer overflows</u>**

Memory Safe Systems Programming

Idea 5: Choose your Tools & Languages Carefully

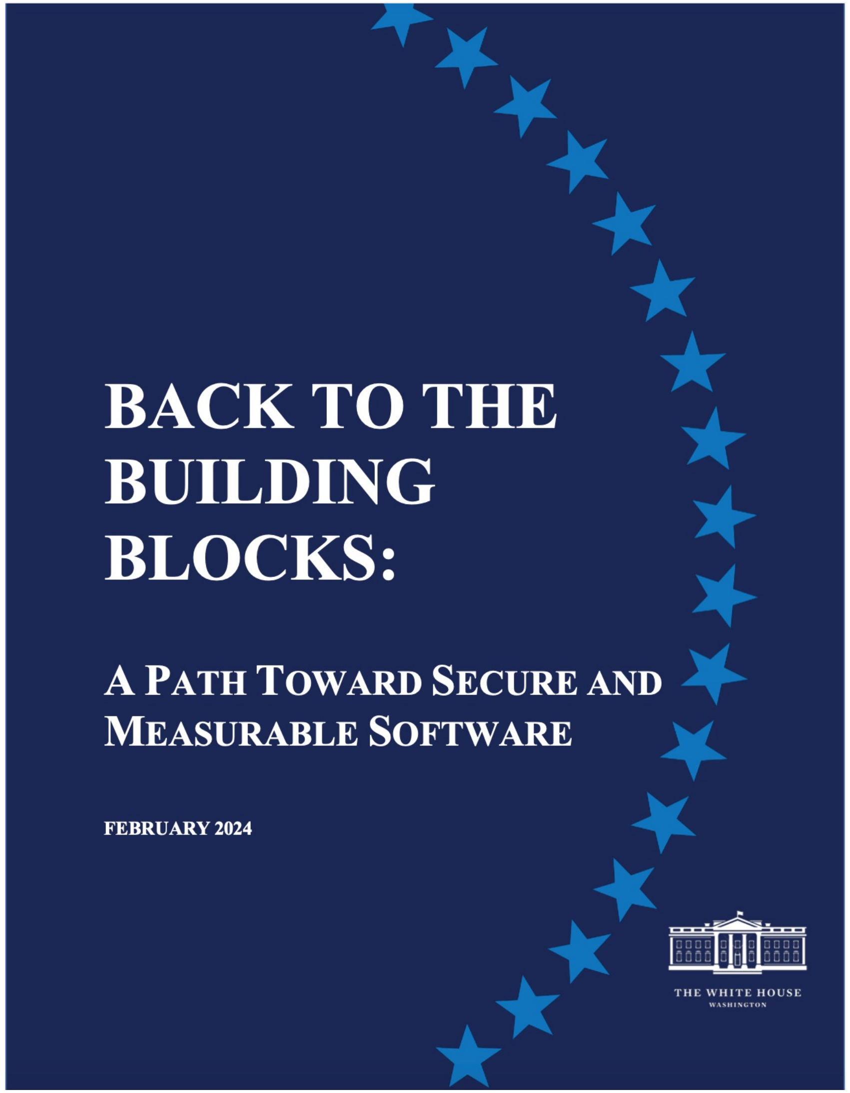
Existing code bases or requirements for a project may dictate what tools you use. Knowing C is crucial – it is and will remain widely used.

When you you are choosing tools for systems programming, consider languages that can help guard against programmer error.

- Rust (Mozilla)
- Go (Google)
- Project Verona (Microsoft)

Memory Safe Systems Programming

"Memory safety vulnerabilities are a class of vulnerability affecting how memory can be accessed, written, allocated, or deallocated in unintended ways. Experts have identified a few programming languages that both lack traits associated with memory safety and also have high proliferation across critical systems, such as C and C++. Choosing to use memory safe programming languages at the outset, as recommended by the Cybersecurity and Infrastructure Security Agency's (CISA) Open-Source Software Security Roadmap is one example of developing software in a secure-by-design manner."



Association for Computing Machinery (ACM) Code of Ethics

ACM Code of Ethics and Professional Conduct

ACM Code of Ethics and Professional Conduct

Preamble

Computing professionals' actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good. The ACM Code of Ethics and Professional Conduct ("the Code") expresses the conscience of the profession.

The Code is designed to inspire and guide the ethical conduct of all computing professionals, including current and aspiring practitioners, instructors, students, influencers, and anyone who uses computing technology in an impactful way. Additionally, the Code serves as a basis for remediation when violations occur. The Code includes principles formulated as statements of responsibility, based on the understanding that the public good is always the primary consideration. Each principle is supplemented by guidelines, which provide explanations to assist computing professionals in understanding and applying the principle.

On This Pa

https://www.acm.org/code-of-ethics

ACM Code of Ethics on Security

2.9 Design and implement systems that are robustly and usably secure.

Breaches of computer security cause harm. Robust security should be a primary consideration when designing and implementing systems. Computing professionals should perform due diligence to ensure the system functions as intended, and take appropriate action to secure resources against accidental and intentional misuse, modification, and denial of service. As threats can arise and change after a system is deployed, computing professionals should integrate mitigation techniques and policies, such as monitoring, patching, and vulnerability reporting. Computing professionals should also take steps to ensure parties affected by data breaches are notified in a timely and clear manner, providing appropriate guidance and remediation.

To ensure the system achieves its intended purpose, security features should be designed to be as intuitive and easy to use as possible. Computing professionals should discourage security precautions that are too confusing, are situationally inappropriate, or otherwise inhibit legitimate use.

In cases where misuse or harm are predictable or unavoidable, the best option may be to not implement the system.

https://www.acm.org/code-of-ethics

Buffer Overflows

- We must always ensure that memory operations we perform don’t improperly read or write memory.
  - E.g. don’t copy a string into a space that is too small!
  - E.g. don’t ask for the string length of an uninitialized string!
- The **Valgrind** tool may be able to help track down memory-related issues.
  - See cs107.stanford.edu/resources/valgrind
  - We’ll talk about Valgrind more when we talk about dynamically-allocated memory.
  - Valgrind can detect some, but not all, stack-memory-related issues

Recap

- **Recap:** Strings so far
- Searching in Strings
- **Practice:** Password Verification
- Buffer Overflows and Security

Lecture 7 takeaway: string searching functions allow us to search for characters, substrings, and spans. Buffer overflows can cause significant functionality and security issues!

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

Extra Practice

2. Code study: strncpy

STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, **strncpy**() writes additional null bytes to dest to ensure that a total of n bytes are written.

A simple implementation of **strncpy**() might be:

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
       dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

What happens if we call `strncpy(buf, str, 5);`?

2. Code study: strncpy

STRCPY(3) Linux Programmer's Manual STRCPY(3)

DESCRIPTION

The **strncpy**() function is similar, except that at most n bytes of src are copied. **Warning:** If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

If the length of src is less than n, **strncpy**() writes additional null bytes to dest to ensure that a total of n bytes are written.

A simple implementation of **strncpy**() might be:

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

dest	
src	
n	5
i	

What happens if we call `strncpy(buf, str, 5);`?