CS 107
Lecture 17: development process / testing / git

Friday, March 9, 2018

Computer Systems
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg
Today's Topics

• Programs from class: /afs/ir/class/cs107/samples/lect17
• A bit more on memory throughput / latency
• The Apple MacOS High Sierra root login bug
  • The exploit
  • The fix (forced!)
  • The remaining problems
• Development Process / Testing your code
  • Let's build a simple encryption program.
• git
  • Why Version Control?
  • What does git provide?
  • You already know the clone command
  • Committing
  • Reverting
  • Github
Memory Throughput -vs- Memory Latency

• On Monday we saw the following diagram, and the question about throughput and latency came up.

• Latency is the amount of time it takes for a single memory access, which could be one or more bytes. In the diagram above, it takes three cycles for a memory request for a register, and millions of cycles for the disk.

• Throughput is the amount of bytes you can continually read. Above, you can read 16 bytes per cycle for registers, but only 1 byte every 30 cycles for the disk.
Latency Numbers Every Programmer Should Know

- 1 ns
- L1 cache reference: 0.5 ns
- Branch mispredict: 5 ns
- L2 cache reference: 7 ns
- Mutex lock/unlock: 25 ns

- Main memory reference: 100 ns
  - Compress 1KB with Zippy: 3 µs
  - Result: 1 µs

- Send 1KB over 1Gbps network: 10 µs
- SSD random read (1Gb/s SSD): 150 µs
- Read 1 MB sequentially from memory: 250 µs
- Round trip in same datacenter: 500 µs
  - Result: 1 ms

- Read 1 MB sequentially from SSD: 1 ms
- Disk seek: 10 ms
- Packet roundtrip CA to Netherlands: 150 ms

Source: https://stackoverflow.com/a/4087315/561677
Testing Your Code
The Apple MacOS High Sierra root login bug

• Apple recently upgraded its operating system.
• But, there was a problem...

Lemi Orhan Ergin
@lemiorhan

Dear @AppleSupport, we noticed a *HUGE* security issue at MacOS High Sierra. Anyone can login as "root" with empty password after clicking on login button several times. Are you aware of it @Apple?

10:38 AM - 28 Nov 2017

https://objective-see.com/blog/blog_0x24.html
Companies like Apple aren't supposed to have bugs that let anyone log into your computer as root...what happened?

Basically (and read the excellent blog post for details):

- When a user tries to log in with a "disabled" account, the system will create a password for that account with whatever password the user typed in (and the root account on MacOS is disabled by default, but exists).
- The first login doesn't work, but subsequent logins work just fine, because now the password has been created.
- Oops! Why would a password be created for a disabled account for a user that is simply attempting to log in?
  - If a legitimate user's password has an old encryption method from a previous OS version, Apple (ironically) upgrades to a newer encryption standard.
  - But, the code ignores a failed login attempt and upgrades the password with the one the user typed in, storing it into the account…
The Apple MacOS High Sierra root login bug

- The blog post is full of cool, you-can-understand-this-after-binary-bomb debugger disassemblies (in \texttt{lldb}, a slightly different debugger than \texttt{gdb}):
• How could Apple have found this bug?
• Testing!
  • You have been using `sanitycheck` to test your code for CS 107, and the `test_harness` code for assignment 7 is handy for finding bugs and for determining correctness. But, for production code, many companies have extensive test suites that get run whenever a change is made to the codebase, and the tests are meant to prevent bugs entering into the system.
  • Let's spend some time looking at various ways to effectively test your code.
Testing Your Code

• When you are writing a program or application, you can test for various things (this is not a comprehensive list):

  • **correctness** — does it do what it is supposed to do? This is where `sanitycheck` and `test_harness` come into play.
  • **efficiency/performance** — where are the bottlenecks? Use `calgrind` and other tools to analyze your code for efficiencies.
  • **security** — this ties into correctness, but can be more subtle (e.g., if you only test on correct input, you may miss many security issues).
  • **memory leaks / errors** — this can be a security issue, but it can also cause slowdowns and over-use of memory.
  • **scalability** — this is related to performance
Testing Your Code

• How can we run our tests?

• **black box testing** — you don't know (or don't care) about the implementation: testing is done based on the interface
  • can be done internally in the program, or externally by running the program (e.g., `test_harness` vs `sanitycheck`)

• **white box / glass box testing** — you *do* know the internal structure of the code, and you can test it at a lower level, to get good "code coverage"
  • **unit tests** — low-level tests that are simple and can ensure that all code is covered
  • **system tests** — test all the pieces together (can find interactions between parts of your program that unit tests cannot)

• As the programmer, you should run your own tests! Relying on a Quality Assurance (QA) team will annoy them and is counter-productive.
Let's Test Some Code

• Let's test the **one_time_pad_buggy** program in /afs/ir/class/cs107/samples/lect17
• A "one time pad" is an encryption method that takes an input text (or binary) and a key text (or binary) and **XORs** characters from the input with characters from the key to produce output characters, which are now encrypted.
• To decrypt text, perform the exact same XOR method with the encrypted text as input and the same key file.

• The usage for the program is as follows:

  ```bash
  ./one_time_pad_buggy input_filename key_filename key_offset output_filename
  ```

• The offset is where we want to start reading from the key file.
• At this point (without looking at the code), what kind of testing can we do?  

  Black Box
• First, we should try some easy input.
• Let's create two files:
  • `input1.txt`: "This is some text to encrypt."
  • `key1.txt`: "We will use this as the key text."
• Run the code:
  
  $ ./one_time_pad_buggy test_files/input1.txt test_files/key1.txt 0 output1.bin
  
  Performed cipher on text. New offset: 30
• If we look at the output file, it will look like gibberish, but we can see if we can decrypt:
  
  $ ./one_time_pad_buggy output1.bin test_files/key1.txt 0 unencrypted1.txt
  
  Performed cipher on text. New offset: 30
• $ diff test_files/input1.txt unencrypted1.txt shows no differences.
• Great! Let's put the test into a shell file (this is above and beyond the scope of the class, but it is something that you might want to learn on your own).
#!/bin/bash

# report on test function
report_test()
{
    if [ $? -ne 0 ]; then
        printf "$1 failed!\n\n"
    else
        printf "$1 passed!\n\n"
    fi
}

if [ $# -eq 0 ]; then
    printf "Usage:\n\t./`basename "$0"` test_program\n"
    exit -1
fi

program=$1

# Test 1: small input files
./${program} test_files/input1.txt test_files/key1.txt 0 output1.bin
./${program} output1.bin test_files/key1.txt 0 unencrypted1.txt
diff test_files/input1.txt unencrypted1.txt

report_test "test 1 (small input)"

# clean up
rm -f unencrypted1.txt output1.bin
Let's write some more tests. What kinds of input/output can we have that might put some pressure on the program?

- key file doesn't have enough characters
- large files (MB / GB?)
- nulls in the input or key?
- 0-length input file?
- filenames that aren't real files?
- negative offset?
- memory issues?
- Lots of potential tests to do! (let's look at some of them -- some might not report a failure, but you can tell from the program output!)

/afs/ir/class/cs107/samples/lect17/test_files/black_box.sh
Let's look at the code

You can look at the code on the handout or in `/afs/ir/class/cs107/samples/lect17/one_time_pad.c`

Now that you see the code, you can either see the errors directly, or do "white box testing," where you can write tests that "cover" the code -- in other words, you write tests that run on as much of the code as you can.

For example, we can write a test that explicitly tests the cipher function:

```c
// perform encryption/decryption on input using the key at the offset
void cipher(struct bytearray *input, const struct bytearray *keychars,
        size_t offset) {
    for (size_t i = 0; i < input->length; i++) {
        input->bytes[i] ^= keychars->bytes[i+offset];
    }
}
```
Simple test to see if cipher is working

```c
// perform encryption/decryption on input using the key at the offset
void cipher(struct bytearray *input, const struct bytearray *keychars, size_t offset) {
    for (size_t i = 0; i < input->length; i++) {
        input->bytes[i] ^= keychars->bytes[i+offset];
    }
}

// test cipher function
void test_cipher1()
{
    struct bytearray input, key;
    char input_str[] = "a";

    input.bytes = input_str;
    input.length = 1;

    key.bytes = "b";
    key.length = 1;

    cipher(&input, &key, 0);
    // we know that a ^ b should be 3
    // because 97 ^ 98 == 3 (using ASCII values)
    assert(input.bytes[0] == 3);
}
```
A more robust test to see if cipher is working

```c
void test_cipher2()
{
    struct bytearray input, key;
    char input_str[] = "This is some text";
    char *original = strdup(input_str);

    input.bytes = input_str;
    input.length = strlen(input.bytes);

    key.bytes = "A key that is longer than the input";
    key.length = strlen(key.bytes);

    cipher(&input, &key, 0); // encrypts
    cipher(&input, &key, 0); // decrypts

    assert(strcmp(original, input.bytes) == 0);
    free(original);
}
```
In case of fire

1. git commit
2. git push
3. leave building
You may have noticed that you have to use git every time you set up a project in your home directory. The command looks like this:

```bash
git clone /afs/ir/class/cs107/repos/assign7/$USER assign7
```

But, what is this git program, anyway??
git is a *version control system* (VCS) that tracks changes in a directory on your computer (in our case, the myth machines).

Version control systems are useful so you don't end up with directories like this:
git was created in 2005 by Linus Torvalds, who is more famous for... creating Linux!
git was created in 2005 by Linus Torvalds, who is more famous for...creating Linux!

Torvalds was not happy with other version control systems, so he went and created his own with the features he wanted (also, the one the Linux kernel developers had been using became a paid application)

He had some main goals for the project:

• Fix poor workflows from other VCSs
• Include very good safeguards against corruption -- in other words, make it bulletproof.
All VCSs ostensibly enable different people to work on a common codebase with minimal problems with sharing code. You've only been using it for yourself (and we use your repositories when you submit assignments), but you could have multiple people working at the same time on the same codebase (generally in different files).

git is "distributed" in that it keeps an entire history of a repository in every user's folders. This makes it robust -- if you have a cloned copy, you have all the history, forever for that code.

git is scalable, and works with giant projects (e.g., Google has migrated much of their code to git)
As you have noticed, the "git clone" command puts a copy of a repository into a folder of your choosing. But what else can it do?

You can "commit" changes. E.g.,

```
$ git commit -m "finished up realloc for implicit heap allocator" implicit.c
[master 4756990] finished up realloc for implicit heap allocator
  1 file changed, 1 insertion(+), 1 deletion(-)
```

This means, "I want you to keep track of the current state of implicit.c, and this is a message about what changes I made (-m).

When you commit changes, the changes are there forever (well, in your own directory, anyway)! You can always go back to a previous version, whenever you want.
To see the log of your git history, you use the "git log" command:

```bash
$ git log
commit 4756990e3443742ec586ed1bf66b55d74384a37c
Author: Chris Gregg <tofergregg@gmail.com>
Date: Fri Dec 1 12:22:57 2017 -0800
  finished up realloc for implicit heap allocator

commit 091934cb6b99f0fe0f457db6464fd0da430512d0
Author: CS107 tools <cs107@cs.stanford.edu>
Date: Wed Nov 29 00:26:30 2017 -0800
  Created starter 17-1 assign7 cgregg

commit 18ec7d3008bd6011a2dd090a4cc4d4df95953ec9
Author: CS107 tools <cs107@cs.stanford.edu>
Date: Wed Nov 29 00:26:30 2017 -0800
  Init empty repo
```
To see the log of your git history, you use the "git log" command:

```
$ git log
commit 4756990e3443742ec586ed1bf66b55d74384a37c
Author: Chris Gregg <tofergregg@gmail.com>
Date:   Fri Dec 1 12:22:57 2017 -0800

    finished up realloc for implicit heap allocator
```

```
commit 091934cb6b99f0fe0f457db6464fd0da430512d0
Author: CS107 tools <cs107@cs.stanford.edu>
Date:   Wed Nov 29 00:26:30 2017 -0800

    Created starter 17-1 assign7 cgregg
```

```
commit 18ec7d3008bd6011a2dd090a4cc4d4df95953ec9
Author: CS107 tools <cs107@cs.stanford.edu>
Date:   Wed Nov 29 00:26:30 2017 -0800

    Init empty repo
```
The "SHA1" cryptographic hash produces a 160-bit (40-digit) hash from an input (a text file, program, etc.)

A cryptographic hash has a property that it produces a bit string of a fixed size that is infeasible to reverse -- it is a "one-way function". In other words, it is very difficult (or impossible) to go backwards from "5deb7cde8c28ddb379a634d11ff16c52d115d74d" to the program text it came from (or any part of it). A second property is that it is also infeasible to find another text that produces the same hash.

The standard is a bit old now (released in 1993), and it has been successfully attacked: see https://en.wikipedia.org/wiki/SHA-1#Cryptanalysis_and_validation

```bash
$ shasum one_time_pad
0165b74d3e93f958ceae8e760b94b8d235ca5c3c  one_time_pad
$
```
The latest attacks, "the SHAppening" and "SHAttered" have shown that the integrity of SHA-1 is not good enough to be reliable for extremely sensitive data.

One use of cryptographic hashes is to "sign" a document or a set of data, to prove that the data is the same as the creator intended. For example, if you create a binary file (say, `one_time_pad`) and provide the hash to someone along with the binary, the other person can also run `shasum` on the binary, and if the hash is the same, they should be able to guarantee that the file hasn’t been changed since you created it (otherwise, it could have been hacked!).

But, SHAttered showed that it is feasible to create another file with the same hash...bad news!
You can go back to your old versions any time, with the hash:

```
$ git checkout 091934cb6b99f0fe0f457db6464fd0da430512d0
Note: checking out '091934cb6b99f0fe0f457db6464fd0da430512d0'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using \(-b\) with the checkout command again. Example:

```
git checkout -b new_branch_name
```

HEAD is now at 091934c... Created starter 17-1 assign7 cgregg

Once you are finished looking around (for instance), you can get back to the original:

```
$ git checkout master
```
If you do change things while you have a version checked out, you might see a message like this:

```
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
   implicit.c
Please, commit your changes or stash them before you can switch branches.
Aborting
```

Scary! One of the things Niels is going to talk about is why this doesn't need to be scary.

There are options here, but the bottom line is that you should know when to work on changes, and when not to work on them. :)
If you have cloned a repository, there are now two full copies of the repository. If you want to make them the same (i.e., to keep them consistent), you can "push" your changes into the other repository (you have to own both of them, and you own the one you've cloned from on myth machines):

```
$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 315 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To /afs/ir/class/cs107/repos/assign7/cgregg
  091934c..4756990  master -> master
```

Now the changes are stored in both places. If there were changes that you (or others) made to a different repository, you can "pull" those changes into your own repository copy. Git will do its best to merge the files, but sometimes it needs a bit of help if two people made changes that would overlap.
So, basically, a git workflow might look like this:

1. Create an initial git repository (using `git init`), either on your own computer, or on a server (which can live in another part of the same computer).
2. Clone the repository to where you want to work on the files (if that is a different location than where you created it).
3. Work on the repository, making commits as you change things. We automatically commit and push your repositories every time you use `sanitycheck` or `submit`. But, you can always commit more often.
4. Push your repositories when you want to keep them backed up to a different location, or just keep them in sync.
5. Pull from the repository if there are changes you want in your local copy.
References and Advanced Reading

• References:
  • CS190 lecture on testing: https://web.stanford.edu/~ouster/cgi-bin/cs190-spring15/lecture.php?topic=testing
  • One time pad: https://en.wikipedia.org/wiki/One-time_pad

• Advanced Reading:
  • Performance testing: https://stackify.com/ultimate-guide-performance-testing-and-software-testing/
  • Testing 101: https://stackify.com/software-testing-tips/
Extra Slides