The 2015 iCTF Service Format

The rumors are true! This year's iCTF (codename: "crowdsourced evil") is based on services that are submitted by the participating teams. By having the teams create services, we can experiment with a new competition design that, in addition to testing attack and defense skills, tests the ability of each team to write vulnerable software. Creating a service that is fun to exploit (and can be exploited in the intended way only) requires creativity and attention to details. In addition, the service should not be too hard or too easy, as this will affect the score of the team who submitted the service.

The services are due on Sunday Nov 29th, 2015 at 11:59pm. If you do not submit a service by then, you will not be able to participate in the competition (sorry, but we need to be strict with this, so that we have the time to test the various services and their possible interactions). Of course you also have to register for the competition as it was done in previous years.

This iCTF is the first competition of this kind, and therefore we are sure that SOMETHING WILL GO WRONG. However, in order to try to eliminate some of the thousands of unforeseeable issues, we have put together a detailed description of how you should package your service.

Your service can do ANYTHING YOU WANT™ (well, almost…), but in some way it must be designed for people to store their "secrets" (that is the flags) and hope that they can be read only if you have their real credentials. Of course, your service also needs to have one vulnerability. And the vulnerability can be… yes, ANYTHING YOU WANT™. However, be careful as something that is too easy or too difficult will hurt you.

We are bracing to receive services that are fun, crazy, different and maybe something that gives us an 'awesome!' moment. Maybe something that teaches us how to do (or not do…) things. You have a month to put this together… Start early and be done!

The service must follow a precise format, which is described in this very long document. We put this document together to get most questions out of the way before you start writing. However, feel free to ask question by sending an email to ctf-admin@lists.cs.ucsb.edu.

First, here are a few important things to remember:

- You can find examples of valid services in service_samples. If you feel lost it is probably best to start by modifying those.
- Teams will have to run a VM with many services. Not everyone has a supercomputer… keep your service light on CPU and memory. Same goes for the network: we are not Google, and probably neither are you :)
- For even more details, and the rationale behind some choices, you can refer to our paper: http://ictf.cs.ucsb.edu/pdfs/2014_3GSE_iCTF.pdf
- We're not covering scoring details here. Keep an eye on the mailing list for that.

SERVICE DESIGN

A service is a program that, from a high-level point of view, has a stated benign purpose (e.g., it implements a web forum) and contains some flags: secrets that are not supposed to be revealed to unauthorized users during normal operation (e.g., private messages in the web forum).

The goal of the opponent teams is to steal a flag from the service and submit it to the organizers. Services are written with a variety of deliberately incorporated vulnerabilities, so that stealing flags is indeed possible, unless the defending team is able to patch all vulnerabilities successfully.

Generally speaking, a service listens on a given port, so that external programs can interact with it and invoke certain functions. This functionality should cover at least the following high-level categories: a *setflag* functionality that allows one to set the currently active flag; a *getflag* functionality that allows the benign and authorized retrieval of a flag; some additional benign functionality that represents the normal behavior of the service.

For instance, a service might implement a simple social network, where it is possible to create a user (with a given username and password) and to set a status. In this example, the user's status is the secret flag, and the status can only be read if the user is logged in by providing his username and password. Here, the *setflag* functionality would take as input a status to be set, and it would then pick (or create) a user, login and update the status, and save the chosen username and password for future reference; the *getflag* functionality would take as input those credentials and retrieve the user's status after logging in. Note that the password is only known to the organizer and acts as an authentication token. On the other hand, to steal a flag, the attacker needs to discover and exploit a vulnerability in the service in order to read the status of a certain user whose password is not known.

A new flag is periodically set and retrieved by our scoring infrastructure so that, at any point in time, each service has one and only one active flag. The scorebot does so by using the *setflag* and *getflag* functionality. The *setflag* and *getflag* functionality should blend with benign service interactions and not stick out in an obvious way. Not only the *setflag* and *getflag* should resemble benign interactions, but benign interactions should resemble *setflag* and *getflag* functionalities as well. In fact, the *setflag* and *getflag* functionalities can be used for benign interactions by simply setting or retrieving inactive or incorrect flags.

In particular, the *setflag* script takes as input a *flag*, and is in charge of installing it into the service and returning the corresponding *flag_id*,and *token*. The *getflag* script takes as input the flag_id and its *token* and should retrieve the corresponding *flag*, so that the scoring infrastructure can determine whether the service is properly functioning or not. Note that the *getflag* script can retrieve the *flag* associated to a given *flag_id* because it has access to its associated *token*, and **not** by exploiting a vulnerability.

The *benign* script exercises other service-specific benign functionality, so that the *setflag* and *getflag* scripts do not stick out in any obvious way, and to ensure that the service is fully operational and has not been tampered with (e.g., because of a careless patching process).

Finally, the *exploit* script is used to demonstrate one possible working exploit against the service. Of course, the teams will develop many more during the competition, but one must be included with the service submission. Conceptually, an *exploit* script needs to steal the *flag* associated with a given *flag_id*, without

knowing the associated *token*. Note that the *flag_id* specifies which of the many flags of the service the script must retrieve. For this reason, the only input the *exploit* script takes is the *flag_id*, and it must return the *flag*.

SERVICE FORMAT

The service must be submitted as a single tar gz file (.tgz extension). Run 'make' in the sample service directories to get a sample. Hereinafter, we describe the contents of the various files that need to be present in the bundle.

info.yaml:

Once decompressed a service bundle must contain a file named `info.yaml`.
It must define a dictionary with the following mandatory fields:

- `service_name`: the name of the service (the length must be between 4 and 32 characters, only alphanumeric characters and "-_:" are allowed).
- `type`: either "console" or "web". A console service will be run using xinetd, whereas a web service will be run in Apache (more details below).
- `description`: a "player-friendly" description of the service. What is the (uncompromised) service supposed to do? Is it a forum? An email server? A teapot controller?
- `flag_id_description`: a "player-friendly" description of what the `flag_id` is in this service. That is, how do we tell "secrets" apart from one another? Your service must store many, but we set or get one flag at a time. Same goes for exploiters: we tell them which flag to get, and accept only that one. More details on this later.

Services of type "console" must also contain the following field:

- `architecture`: one of "x86_64","i386","mips","aarch64","ppc","mipsel","armeb","arm". This value determines the architecture on which they service runs. In case the architecture is different than "x86_64" or "i386", qemu-<architeture_name> will be used to run the binary.
-

These fields are also allowed:

- `apt_dependencies`: a list of debian packages which will be installed before the service installation using the "`apt-get install`" command.
- `pip_dependecies`: a list of pip-installable packages which will be installed before the service installation using "`pip install --user`"

Service files:

Once decompressed a service must also contain the following folders: "`scripts`", "`service`", "`src`".

The `src` folder must contain the source code of the service. Note that this code will *not* be used to recompile any binary file used by the service and will *not* be deployed on the teams' virtual machines. However, it will be used by the organizers for manual verification of the service, especially if problems arise during the competition. Comments are appreciated :)

The `scripts` folder must contain the following files: `benign.py`, `exploit.py`, `setflag.py`, and `getflag.py`. Note that these files will *not* be deployed on the teams' virtual machines.
If you need them, you can import *pwntool*, *requests*, *Crypto*, and *pexpect*. Note that you will *not* have your service's apt or pip dependencies installed, but you can put other python files (even entire libraries, if you want) in your bundle and import them.

Each file should contain the corresponding function:

```
def set_flag(ip, port, flag):
    return { 'FLAG_ID': <generate_a_unique_one>, 'TOKEN': <benign_access_token> }
def get_flag(ip, port, flag_id, token):
    return { 'FLAG': <flag_retrieved_using_the_token> }
def exploit(ip, port, flag_id):
    return { 'FLAG': <flag_retrieved_using_a_vulnerability> }
def benign(ip, port):
     # Nothing to return :)
```

In case of error (the service is not behaving as it should), throw an exception.
Try to be robust, services may not answer immediately if heavily loaded. You may find pwntools or pexpect useful. Again, refer to the introduction and to the samples for more details.

The `service` folder contains the files that will be deployed on the teams' virtual machines: see the next section.

SERVICE DEPLOYMENT

Services live in "`/opt/ctf/<service_name>`" (later referred to as <*service_home*>), as a user named "*ctf_<service_name>*" (belonging to the group "*ctf_<service_name>*", and with home set to *service_home*).
The permissions of the folder <service_home> are: root:root 755.

<service_home> is populated with files from the the `service` folder from your tgz bundle. Only these directories are allowed: "`service/ro`", "`service/rw`", and "`service/www`".
To be specific:

- In a "console" service the `service` folder must contain a folder named "`ro`", which must contain an executable file with the same name of the service. Instead, for a "web" service the `service` folder must contain a folder named "`www`".
- The `service` folder must also contain a folder named "`rw`".
- The submitted service cannot contain symbolic links or .pyc files.

Permissions of these folders are set recursively in the following way:

- `ro` and `www`: ctf:ctf_<service_name> 750
- `rw`: ctf:ctf_<service_name> 770

The idea is that a non-root player (*ctf* user) must be able to patch the service, whereas even a fully exploited service should not "tilt" the game too much. At least, that's what we hope :)
Pip dependencies are installed in `<service_home>/.local` (same permissions as `ro`).

Also: after the installation, if the `ro` folder contains a file named `postinst`, this file is executed as the user *ctf_<service_name>* after cd-ing to `<service_home>/rw`. After its execution, the `postinst` file is deleted.

SERVICE EXECUTION

The teams' virtual machine will run an updated Ubuntu 14.04 64-bit server distribution.
In addition to the packages available by default in this distribution, the following packages are installed:
'`libapache2-mpm-itk`', '`curl`', '`apache2`', '`apache2-bin`', '`apache2-mpm-itk`',

‘libapache2-mod-php5’, 'gdebi', 'xinetd’, ‘qemu-user-static’,‘qemu-user’,‘python-pip’, ‘sqlite3’, ‘libsqlite3-dev'.
Access to `/proc` and `dmesg` is limited to avoid cross-service information disclosure.

“console” services are run once for each connection, and communicate via stdin and stdout.
In particular, they are executed using xinetd with the following configuration:

```
{
    socket_type = stream
    protocol    = tcp
    wait        = no
    user        = ctf_<service_name>
    bind        = 0.0.0.0
    server      = <service_home>/ro/wrapper.sh
    port        = <a port selected during the virtual machine generation>
    type        = UNLISTED
    instances   = 50
}
```

`wrapper.sh` is an automatically-generated script that takes care of:

- Running the file `<service_home>/ro/<service_name>` (using the right qemu version if required, according to value of the `architecture` field in `info.yaml`).
- `chdir` to the `<service_home>/rw/` folder.
- redirect stderr to `/dev/null`

Instead, `apache2` (`apache2-mpm-itk` enforces the user-level separation of the different services) serves the www directory of “web” services.
In particular, the following configuration is used (`/etc/apache2/sites-enabled/<service_name>.conf`):

```
Listen <a port selected during the virtual machine generation>
<VirtualHost *:<that_port> >
    DocumentRoot <service_home>/www/
    AssignUserID ctf_<service_name> ctf_<service_name>
    AddHandler cgi-script .cgi
    <Directory <service_home>/www/>
        Options ExecCGI FollowSymlinks
        AllowOverride None
        Require all granted
    </Directory>
    ErrorLog <service_home>/rw/error.log
    CustomLog <service_home>/rw/access.log combined
</VirtualHost>
```

You can structure your `www` directory as you wish, but keep in mind that it will be read-only. Same principle as “console” services.
Beside regular files, you can use:

- `<something>.php` files will be handled by libapache2-mod-php
- `<something>.cgi` executable files will be treated as CGI programs (see https://httpd.apache.org/docs/howto/cgi.html)

WHERE TO STORE FLAGS

In short: in the `rw` directory, but don't trust it too much.

To be specific: we `cd` to your `<service_home>/rw` directory before running your program, and that's the only directory your service can write to. Storage details are up to you, but keep in mind that:

- Multiple processes can run at the same time.
- Some of them may be exploited.

If in doubt, keep it simple: one file per flag, named as the `flag_id`, which you should make long and random.
For SQLite refer to https://www.sqlite.org/faq.html#q5.

Thanks for having read through the whole thing ;-)
If you have still questions, please send a message to ctf-admin@lists.cs.ucsb.edu.
We will maintain a FAQ page with the questions that we can answer publicly.