SEE TEXT ONLY

GAMING BSD

IF YOU HAVE BEGUN READING THIS, IT PROBABLY MEANS YOU ARE INTERESTED IN SECURITY OR IN EMBEDDED SOFTWARE OR YOU WORK IN THE GAMING BUSINESS. IN ANY CASE, PLEASE PUT ASIDE YOUR ANTIPSYCHOTIC MEDICATION RIGHT NOW, BECAUSE WE NEED YOU IN YOUR MOST PARANOID STATE, READY TO WEAR THE TWO HATS OF SECURITY, THE BLACK ONE AND THE WHITE ONE.

SECURITY has been discussed a lot over the years, particularly when focusing on protecting customers' data. That data can be simple data, such as an email address or a user-name, or more elaborate data like a password used to decrypt the shopping done by a customer in an online shop. These services are usually based on data protection and do not ensure that the hardware or software being used is what is actually delivered. In the case of servers, it is useless to talk about hardware protection because only the system administrator should have physical access to those machines.

In the gaming industry, we have software and hardware, both of which must be checked to ensure that the delivered product has not been manipulated. For example, manipulation could be done by changing some of the product's hardware components or by inserting a way to bypass the protection mechanism implemented on it.

In addition to a hardware intrusion detection system, it is highly recommended that the software developed for the product also have some kind of protection. FreeBSD offers several tools for that, starting with encrypting the disk on

By Roberto Fernández

which the programs are installed and ending by detecting whether they were somehow changed.

We should not forget that not every single piece of code will be used in a gaming system. For instance, `freebsd-update(8)` could be used or not. If you are not using something, it could be a security risk for the system if it is left on a release version. If you do use it, then you have to control which security issues affect your software and how patches could be applied to the system.

But there is more to it than just protecting hardware or software. As a gamer, I am only concerned that the data I saved do not get lost. When I save the progress I made on a game, I would hate to see that get lost and require me to redo what I had already done. Imagine that you have finally gotten over a scene that took maybe five hours to get through, and then the saved point was lost. Would you be "happy" or would you want to tear apart your console?

BUILDING IT AS EMBEDDED

The best way to make sure your software has as few security holes as possible is to use YAGNI (You aren't going to need it). Although this is a technique for writing high-quality code, it could be used here to inform you if a program or a library is not going to be used and thus keep you from including it.

To get a better understanding of how this is accomplished, the building section is divided into three parts: building the world, building the kernel, and building ports.

Building the World

There are two ways to accomplish this:

- Build everything and install something.
- Build only what you need.

The first axiom implies a full build with well-known commands:

```
root@ASUS-R752L: /usr/src # make buildworld
root@ASUS-R752L: /usr/src # make installworld
     DESTDIR=/work/target
```

And after its completion, all the items go into the final system with a shell script. This is a way to start if you do not want to mess around with the FreeBSD build process. An optimization technique for this process is to write a `src.conf(5)` file and set the SRCCONF variable in your environment. By doing that, your build process could boost up a lot, depending on how many of the settings are left out of the build process.

The second axiom requires the development of a build variable like the ones from `src.conf(5)`. Explaining this fully would require an entire article in and of itself, and it is not the main focus here. With proper implementation of the variable, the build process could be sped up by 50% and thus avoid long waiting times.

Building the Kernel

Inside the kernel configuration file, there are a lot of devices and options that might never get used on your machine. Examples are NFS options or RAID devices. Again, if you do not need them in your kernel, a new kernel's configuration file should be written without them in order to get the kernel to run faster and, in case someone wants to add hardware to the final product, to reduce the security risk if it is not supported.

The kernel has already been configured to run faster and contains support for the devices you want, but that is not all there is to it. The boot partition must not be encrypted (the last time I checked, Allan Jude was not done with the geli boot loader, at least in an automatic way to boot without typing the password) and the `loader.conf(5)` is saved there, which allows an attacker to modify it and get support for the hardware they want.

Building Ports

This is the trickiest question of all. We can build the operating system in any version of FreeBSD without messing around with your own build system. But this does not work with ports. If you want to get a set of programs or libraries from the ports tree and install them into your product, you need to do something else.

There are several techniques for this. The one used to create packages that will be used by the final user is `ports-mgnt/poudriere`. This tool creates a jail and builds the ports there using the same version of the base system. After building the ports and staging them, you should write a script that is able to get only the libraries and binaries you want. A simple way is to write a file like the `pkg-plist` which is on most of the ports and describes which components must be installed on the real system. Then the script mentioned before should read this file, fetch the libraries and binaries, and copy them into your target bintree.

Right now, you have all you need to run your gaming system, but there are a lot of security reports for the programs that might be installed on the product. They should be tracked down to ensure that they do not affect your software and that your product will not be compromised.

WRITING ATOMIC DATA

As explained previously, the data we write on disk must be consistent, which means that the writing must not be in an idle state where the data to be read will be shown as corrupted and no longer usable. There are several causes of data loss. The first, and the most common in the gaming industry, is hard shutdown. This means that the system is not shut down by software, but by hardware which causes the data to not be entirely written on the disk.

FreeBSD has developed a way to ensure that the data is consistent by writing it entirely or not at all. The way to do this is `gjournal(8)`. Usually, it is easier and more comfortable to write the data needed by the user in a non-encrypted device or partition. That way the user can back up saved games and a gamer profile, or copy this information to bring it into another system. Also, the support team can back up user data before starting a reparation.

For this example, we will use a 1-TB hard disk that contains only user data. The GEOM partition scheme should look like the following:

```
root@ASUS-R752L: ~ # gpart show -l ada1
```

```
        34  1953525101  ada1        GPT  (932G)
        34           6        - free -  (3.0K)
        40  1953525096     1  UserData  (931G)
1953525136           5        - free -  (2.5K)
```

In order to put some journaling into the `gpt/User-Data` partition, which is set in a system with 8 GB of physical memory, the following commands must be run:

```
root@ASUS-R752L: ~ # gjournal label -s 16G \
     gpt/UserData
root@ASUS-R752L: ~ # ls /dev/ufs/UserData*
UserData UserData.journal
root@ASUS-R752L: ~ # newfs -J -L UserData
     /dev/gpt/UserData.journal
```

After that, the `/dev/ufs/UserData` entry should be written in the `fstab(5)`. It is important to use that one and not the GEOM identifier or the driver name, because it can cause the wrong partition or device to be mounted.

PROTECTING VITAL DATA

One of the biggest worries in the embedded industry is that proprietary software gets modified or its behavior discovered. This is done to bypass authentication methods implemented to deny the execution of unknown software that allows an attacker the use of modified data or, in the case of the gaming industry, to play an illegal copy of your software.

A simple way to protect your data is to encrypt it with the cryptographic GEOM class, also known as `geli(8)`, where you select the programs to be protected. You can encrypt the whole system or only a partition where your programs are going to be stored, so that in boot time, you can check if any manipulation took place and decrypt this partition if none have been taken. In either case, you have to find a secure place to store your key. It can be generated in runtime or stored in a chip connected to your system, which, under certain circumstances, will give you back the key.

The Trusted Computing Group has developed a standard for this purpose—The Trusted Platform Module or TPM, which is a chip that has, among other things, a non-volatile memory where you can write information and seal it with values stored in the registers. Usually, the BIOS will calculate its own checksum and the MBR's and store them in the TPM's registers. After the system has completely booted, the other registers can be written with some calculated data to have access to the information stored in the chip's memory which holds the key for decrypting the root partition.

If your platform is using a UEFI boot, then the Secure Boot is an additional step that could be used, allowing the machine to have a signed boot process and to look for a manipulation on the boot chain.

AUTHENTICATING PROPRIETARY SOFTWARE

The authentication of proprietary software comes after the machine has completely booted. The simplest way to do this is to write in a file which files must be checked and which signature they must have, the same thing FreeBSD does with release files (installation media).

Another way to do it is to write a program that reads from a binary file using a defined protocol and checks whether the programs were somehow altered. This version requires a thorough understanding of authentication methods and how to calculate checksums of binary files.

But what is interesting right now is not how to detect a manipulation in your software, but what to do when you detect one. As an example, Microsoft® bans the users who tried to use a backed-up copy of a game in a Xbox 360® from the online service. It prohibits the users from updating the system or downloading games using that account. Thus it leads the user to create a new one and replay all the games again in order to restore the user's online status as gamer.

That is ideal, but if what has been altered is how the system works, then it is a good idea to "forget" the key to decrypt the right partition, so that the user will have a real expensive paperweight that will never boot again. It is worthwhile to stop and consider the severity of the manipulation and act accordingly. For example, it is not a good idea to destroy the decryption key when the user tries to play an illegal copy of your games when the worst thing that can be achieved is playing the game. On the other hand, if the manipulation could cause the system to be used as a fraud machine, then that option is the right call.

DETECTING HARDWARE MANIPULATION

When installing the product, information about the system can be obtained and stored on a file that will be read after the system has booted to check whether your hardware has changed. The list of hardware attached to your machine and where it is attached can be obtained by running `pciconf(8)` and `usbconfig(8)`, so you can have control of which components are connected via a PCI card and which ones are attached via USB ports or HUBs.

Let's consider an example. The first of the following scripts will get the information gathered by `pciconf(8)` and `usbconfig(8)` and store it in plain text under `/var/db/hw` with the names `pci.db` and `usb.db`. The second will get the information and compare it with the information stored in those files. To get a better understanding of the process, let's remove a security layer (encryption, authentication, or other such mechanisms) from the file and it will be stored in plain text.

```
#!/bin/sh

[ ! -d /var/db/hw ] && mkdir -p /var/db/hw
pciconf -l | sort | cut -d '@' -f1 |\
   while read DEVICE ; do
      echo ${DEVICE} | grep -q 'none' && continue

      pciconf -lv ${DEVICE}
   done > /var/db/hw/pci.db

usbconfig | cut -d ':' -f1 | sort | \
   while read DEVICE ; do
      usbconfig -d ${DEVICE} dump_device_desc
   done > /var/db/hw/usb.db
---------------------------------------
#!/bin/sh

pciconf -l | sort | cut -d '@' -f1 |\
   while read DEVICE ; do
      echo ${DEVICE} | grep -q 'none' && continue

      pciconf -lv ${DEVICE}
   done > /tmp/pci.db

usbconfig | cut -d ':' -f1 | sort | \
  while read DEVICE ; do
      usbconfig -d ${DEVICE} dump_device_desc
   done > /tmp/usb.db

test "$(comm -3 /tmp/pci.db /var/db/hw/pci.db)"
[ ${?} -eq 0 ] && exit 1
test "$(comm -3 /tmp/usb.db /var/db/hw/usb.db)"
[ ${?} -eq 0 ] && exit 1

exit 0
```

If a TPM chip is installed on your product or the UEFI Secure Boot has been selected, your BIOS has already been checked and therefore there is no need to check it again. The port **sysutils/dmidecode** offers the option of checking the vendor and version of the BIOS or your motherboard if there is a SMBIOS or DMI entry point. If `dmidecode(8)` does not work for you, then you should find a way to read this information as it will be worth it in the long run.

The machine's processor is another thing that is worth checking, and `sysctl(8)` is good enough for this. By checking the node `hw.model`, the model of the CPU is returned, allowing the product to check its own processor and boot when it was not changed at all.

INTERFACES

Up till now, we have talked about product. This section explains the risks in connection with any interface to the external world—for instance, the network card. There are several known variations of attack attempts, especially with regard to communications with the external world. Let's consider them through the interfaces.

Network

Today, in the IoT (Internet of things) era, it is normal to communicate a system to the Internet, and it does not matter if it is a toaster or a server. This is a security risk for your product, and some thought is required before connecting a device to the external world. The question to ask is: why must it be connected to the Internet and which services should be allowed?

The product can be connected to the Internet because there might be an online multi-player version of your game or an update service required by the system to keep it up-to-date with the most secure version of the basic software. If only the first condition is relevant, then the product should forbid the user from using other services. But if the product allows the user to have an HTTP browser, then the browser must be configured to deny pages that may compromise the product.

Another issue is the "man in the middle" problem, which asks how you secure your channel to avoid the communication from being sniffed. Will it use IPsec, SSL, TLS, or are you implementing a new cryptographic layer for your communication?

I cannot really say which is better, because each has advantages and disadvantages. It is the mission of the system developer to evaluate the needs of the final product and to set the most secure configuration.

USB

USBs are used for almost everything today. There are adapters for hard disks, Ethernet or wireless connections, webcams, touch screens, and serial communications. One might think that the USB is secure enough, but it depends on which upper layer protocol you are using. If a program using the `libusb(3)` library or a kernel driver is developed, it must be good enough to avoid memory leaks or code injection.

Others

No two systems are identical, so I am not in a position to say which interfaces should be protected and which not. But be paranoid, go through the drivers that are needed, search for security flaws, try to hack them and improve the drivers. FreeBSD can use the help of system developers working in the gaming industry to make it safer!

ROBERTO FERNÁNDEZ lives in Berlin, Germany, where he has worked in the gaming industry for two years and programmed with FreeBSD for three years. When not spending time with his son and wife, he plays video games.