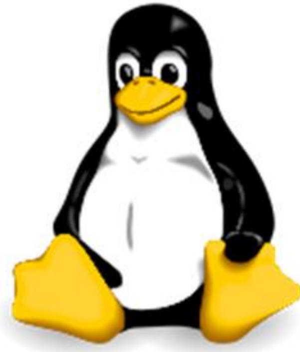
FEABHAS
EMBEDDING SOFTWARE COMPETENCE

Secure is the New Smart

Nick Glynn, Embedded Linux and Technical Consultant at Feabhas shares 25 top tips for hardening the application layer in your Linux device



The Internet of Things (IoT) offers endless possibilities for smart devices and their applications. So it's no wonder that the IoT is as equally tempting to hackers, as it is to developers, keen to showcase their latest developments. A lack of current issues doesn't mean you're OK – you're probably just not being targeted yet!

This paper is designed to help anyone who is developing an internet-enabled Linux device for personal or business use. It highlights the main areas to consider and provides a practical checklist for developing applications for Embedded Linux.

What's the all fuss about?

Linux -based systems are increasingly used in networked devices, as Linux offers a solid POSIX base for API and other conventions, supports a permissions model conducive to a secure system and has industry-wide support.

The ability to create and remotely manage smart devices for utility services, traffic control, or reading meters can have very positive application benefits for business and personal use – however, there are some drawbacks.

High cost of development

From a business perspective, smart devices come at a cost and are much more expensive than their 'dumb' counterparts. For example, the price of a WiFi LED light bulb is almost 50 times the price of a standard LED equivalent (and 500 times the price of a non LED bulb).

To make these smart products attractive despite the wide cost differential, they need to provide either substantial unique consumer benefits (such as unrivalled convenience or even the kudos of being an early adopter) or significant operational cost savings (such as removing the need to take personal meter readings).

Security compromises

The next logical step towards success in the mass-market will be to narrow the price gap. Lower prices will lead to increasing competition and product designers and manufacturers looking for ways to lower the cost of development or improve economies of scale. In some cases, the desire to get to market quickly or cut costs may result in product de-scoping – either of which may adversely affect the attention paid to device security.

© Feabhas 2014

Hackers have already proven that Wi-Fi light bulbs, baby monitors and even pacemakers can be vulnerable to attack. Whilst the roll out of smart meters will enable energy companies to make significant operational cost savings, it is not unthinkable that hackers could find a way to switch all of the meters off – leaving thousands of homeowners and businesses without energy in an all too literal Denial of Service. Not only would the reputational damage be enormous but the costs of addressing the issue would be even more significant than the savings that had been generated. The security breach would need to be identified, solutions determined (for immediate fix and a more permanent solution, if required) and customers would need to be reconnected - as safety standards require each meter to be switched on manually (necessitating an engineer's visit)! At best this may cause a few customers minor issues, at worst, it could cut energy to millions of customers and jeopardise the business.

But it's not just organisations designing and developing devices with embedded systems; many thousands of enthusiasts and students are looking to put their own Linux-based devices online – and they can be just as vulnerable!

Every program is a potential target. Vulnerabilities can be found and used to:

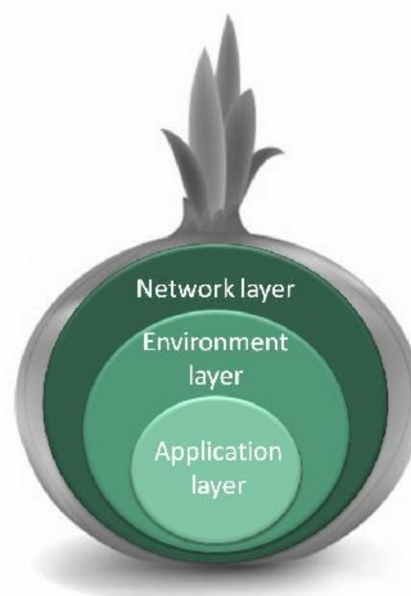
- Crash your software
- Learn your secrets
- Gain control – whether that's to show off or to use your product maliciously

Therefore, it makes sense to build in security from the start.

The Linux security onion

There are various layers that need to be considered in the 'Linux security onion'.

- The network layer – the connected environment such as the internet or IoT
- The environment layer – the Linux operating system
- The application layer – the device's physical system, code and application scripted onto the device by the developer



Securing a device means understanding how and why problems occur and how to address each of these specific layers. For example, C and C++ are not secure languages – they can be subject to format string attacks, buffer overflows or stack and heap overflows - but they are the defacto choice for development on Linux.

Even using a high level language such as Python does not mean that developers can be complacent and assume they are safe from malicious actors. Developers need to take more effective action to secure their devices online.

© Feabhas 2014

The Zen of Hacking

There are a few simple ways that a device can be hacked.

Firstly, it is possible to trick the device into consuming more input than it allocated memory for, to cause a buffer overflow. Once the buffer that lives on the program overflows either:

- the stack is 'smashed'. This allows an overwrite to another stack variable which can be used to take control of the device - often by aiming the CPU at memory you don't control; or
- the heap is corrupted by fooling the system about how it tracks memory. Once corrupted, it is possible to trick the program into writing to arbitrary places in the memory.

Security checklist

The following tips provide a useful checklist for developers wanting to secure the application layer in the Linux security onion.

Authentication and some best practice suggestions

1. Use an authentication mechanism that cannot be bypassed or tampered with. When implementing authentication, ensure that it cannot be bypassed trivially - hardcoded passwords cause issues all the time, as well as "secret" admin pages for web enabled devices where you only need to know the URI
2. Make sure you authorize after you authenticate. Understand the difference between authorisation and authentication and what that means on your platform: Authorisation - What the user can do (Discretionary/Mandatory access controls, read/write access to files etc). Authentication - Who the user is (Username + SSH Keys/Password)
3. Strictly separate data and control instructions, and never process control instructions received from untrusted sources. If you require privileged status to perform functionality - separate the reading/writing of the raw data from the parsing/logic. This prevents bugs and exploits in the data processing side (think XML, JPEG etc) from interfering with the control logic. This is paramount when handling data from unverified sources.
4. Define an approach that ensures all data are explicitly validated and identify sensitive data and how they should be handled. Following on from the previous point - always validate and verify the files entering your system - If you're processing an XML file which consists of a million nested elements, what will happen to your parser? Consider a verification/fuzzing strategy and assume hostile intent!
5. Understand how integrating external components changes your attack surface. The more components added to your system, the larger your attack surface. Think about what happens if you add USB support, do bugs in the USB stack open you up to unexpected strategies? What about userspace applications? Consider these effects when competing in the features race.
6. Be flexible when considering future changes to objects and actors. Take the view that some of the software on your platform will have flaws and it may not always be in the controlled conditions it was originally designed for - always consider an upgrade/patch strategy for your embedded devices
7. Use "safe" string functions. For example, avoid 'strtok' and use 'strtok_r' or 'strtok_s' with -std=c11 instead, in order to prevent buffers being modified or performing 'out of character'

© Feabhas 2014

8. Always know the size of the string and allocate a string large enough to hold the output, including NULL
9. Be wary of NULL and control characters in data you're handling
10. Know the memory model – who allocates, who frees – the caller or the callee?
11. Always allocate enough memory for the expected input and watch out for magic numbers or out of range values!

Architecture and data tips

12. Knowing how your architecture works is fundamental to understanding how it can be used against you – sometimes it can be fun to have a "breakdown session" to see how secure your product is.
13. Shellcode isn't that hard to write... when you know how. Take some time to learn how to at least read it and how it works
14. GDB and objdump are free and highly powerful tools – learn how to use them to understand not what your code **should** do but what it **can** do.
15. New exploit techniques are always being developed – stay on top of them by tracking the CVE lists and ensure you have an update strategy.
16. Always check what data you're being given – eg. gif/jpeg/mp3/wav etc – Do you trust the values given to you? What does your code do when it opens a JPEG that's -100000 by -1000000?

Language, file paths and other coding tips

17. C and C++ are not secure languages so remember to do formal verification when using them – bonus points if it's part of your continuous integration strategy
18. Even understanding how a binary gets into memory in the first place will give you an advantage over other programmers
19. Try not to hard code values - what if you update in one location and not the other?
20. Remember that all command line arguments are in control of the user launching it – are you using getopt or have you rolled your own? Is it secure?
21. Be careful about working with shared files - Who else can read/write to the file?
22. Filepaths can contain .. and ... so be wary of directory traversal attacks
23. Think about file operations. For example, try to avoid API calls that take a path name and prefer those that take a file descriptor instead – this will help mitigate race conditions. And watch out for hard/soft links
24. Don't be afraid to use open-source libraries - Most are under the LGPL which allows dynamic linking without requiring you to open-source your code.
25. Learn what tools are available for your environment – if you aren't willing to discover them, there's a hacker or saboteur, who will!

© Feabhas 2014

Summary - Don’t just ship it - Understand it!

We hope this paper provides some useful insights into why it’s important to secure your device and demonstrates that any device can be vulnerable.

Following these 25 tips for the securing Linux application layer should help you to define an approach that ensures your data is explicitly validated, the software well written and as a result, that you are more confident in the security of your device.

Of course, there is much more a developer can do to ensure that a Linux-based device has greater security online – and essentially, the more you know about your product; the more you can secure it.

If you’d like to learn more about how you can improve the embedded Linux software and security skills for your business or for yourself, please contact us on +44 (0) 1488 73050 or at info@feabhas.com.

“Security is something you have to factor in early – it's not a “bolt on”.

It's your personal reputation and company’s brand on the line.”

References:

1. AVOIDING THE TOP 10 SOFTWARE SECURITY DESIGN FLAWS (IEEE 2014)

© Feabhas 2014

FEABHAS
EMBEDDING SOFTWARE COMPETENCE

About Feabhas

Feabhas improves the competence of embedded software developers through on-site team development, public training for individual engineers, consultancy and mentoring, as well as pre- and post-course assessments.

Feabhas was formed in 1995 and has trained over 15,000 engineers worldwide to date, helping them to improve their embedded software competency and reduce software development times and costs.

As an ARM Approved Training Centre and provider of ARM Accreditation Training, Feabhas is one of only two ARM accredited training partners that offers ARM Accredited Engineer (AAE) programmes in Asia, the Americas and Europe.

Feabhas help with developing software standards e.g. DO-178C, ISO 26262, IEC 62304, EN 50128 etc., graduate training program and re-skilling from other disciplines.

Nick Glynn has a background in embedded software and began his career at Intel. He joined Feabhas in 2012 and is responsible for developing and delivering course material for Linux based training on Kernel/User space and the Android platform in the UK and across the globe. This includes the recently launched “Secure Linux Programming” course.

Nick is also a Consultant on Linux/Android strategies in the embedded space.

For more information about Linux consultancy, programming courses or other security and training requirements, please contact us.

Phone: +44 (0) 1488 73050

E-mail: info@feabhas.com

www.feabhas.com

© Feabhas 2014