Continue

Pwny Documentation Release 0.9.0 Nov 19, 2017 pwnypack is a comprehensive toolkit designed for Capture The Flag (CTF) competitions. It offers a range of command-line utilities and a Python library to facilitate hacking tasks. The pwnypack package provides essential tools for various CTF challenges: 1. pwny: A basic module for general-purpose use. 2. asm: A disassembler for reverse engineering binary files. 3. bytecode: Tools for manipulating Python bytecode. 4. codec: Functions for data transformation and encoding. 5. elf: Utilities for parsing ELF file formats. 6. flow: Communication and network protocol analysis tools. 7. fmtstring: Format string vulnerabilities exploitation module. 8. marshal: Python marshal loader and unploader functions. 9. oracle: Padding oracle attacks and exploitation module. 10. packing: Data (un)packing utilities. 11. php: PHP-related functions for web application hacking. 12. pickle: Pickle tools and serialization utilities. 13. py_internals: In-depth analysis of Python internals. 14. rop: Return-oriented programming (ROP) gadgets generator. 15. shellcode: Shellcode generation module. 16. target: Target definition and exploitation framework. 17. util: General-purpose utility functions. Indices and tables for pwnypack documentation are available in the following sections: 1. Indices and tables 2. Python Module Index pwnypack is officially maintained by Certified Edible Dinosaurs, aiming to provide a comprehensive toolkit for CTF challenges and real-world hacking scenarios. Using Arm Scalable Vector Extension to Optimize OPEN MPI ========== To combat the increasing scale of high-performance computing systems with extension instruction sets, it is essential to utilize various parallelism techniques. The Arm Scalable Vector Extension (SVE) is a vector extension for AArch64 execution mode that supports wide vector extensions and enables automatic adaptation of vector code at runtime. ### Overview of SVE The Armv8 architecture supports various processor architectures, including the recent Armv8-A architecture, which introduces the Scalable Vector Extension (SVE). SVE provides a range of different values that permit vector code to adapt automatically to the current vector length at runtime. This feature enables even greater parallelism and is particularly useful for optimizing memory copy operations. ### Analysis and Performance In this paper, we analyze the usage and performance of SVE instructions in Arm's SVE vector Instruction Set Architecture (ISA). We evaluate the constraints on SVE instructions, which range from a minimum of 128 bits to a maximum of 2048 bits in increments of 128 bits. Our results show that SVE not only takes advantage of using long vectors but also improves performance for various local reduction operations. ### Optimizing OPEN MPI with SVE We utilize the SVE instructions to optimize OPEN MPI, which is a widely used high-performance computing framework. By applying the constraints and optimizing the code for SVE, we achieve significant improvements in performance. ### Conclusion The Arm Scalable Vector Extension (SVE) is a powerful tool for optimizing high-performance computing systems. By utilizing its features and constraints, we can achieve significant improvements in performance. Our work demonstrates the potential of SVE to optimize OPEN MPI and highlights the importance of parallelism techniques in modern computing architectures. Cross-Compiling Linux Kernels on x86 64: a Tutorial on How to Get Started Android uses TrustZone for cryptographic hardware backing. This includes key generation, storage, and validation in the secure world, with non-secure world only getting "tokens." Public keys are accessible in the non-secure world, allowing secret unlocking. NetBSD 9.1 introduces disk encryption using cgd(4) and adds support for Xen 4.13 and ZFS on dk(4) wedges. The update also includes various reliability fixes and improvements, as well as support for USB security keys and more hardware RNGs in the entropy subsystem. Meanwhile, the guide to cross-compiling highlights its importance, especially for embedded systems, new operating systems, or architectures without Nix support. Cross-compilation allows reusing native infrastructure, reducing duplication between package expressions, and specifying target systems through system strings or LLVM triples. Actually, the I686-pc-mingw32 I arm-none-eabi has up to four parts, making it a quadruple rather than a triple. The presence of an optional libc on systems with only one standard Libc is worth noting. Originally, GNU config had just three parts. Some new instructions have been introduced, including a traditional ARM exception model, virtual addresses stored in 32-bit or 64-bit registers depending on AArch64 architecture. New general-purpose registers and instructions such as SIMD, floating-point, and cryptographic functions are also part of this update. #### Architecture Versions The presentation covers various aspects of the AArch64 architecture, including privilege levels, registers, instruction sets, exception models, and memory models. It also explores the concept of a 64-bit Android on ARM. #### Privilege Levels and Registers AArch64 has four exception levels and two security states, with EL0 being the least privileged and EL3 the most privileged. The system includes Secure and Non-Secure states, with Trusted Services operating in both Secure and Normal (Non-Secure) states. #### Exception Model and Memory Model The presentation delves into the details of the AArch64 exception model and memory model, providing an overview of how virtual addresses are stored and managed. #### Table of Contents The guide covers various topics related to iOS pentesting, including setting up an iOS pentest lab, acquiring iOS binaries, generating an iOS binary from Xcode source code, installing iOS binaries on physical devices, guesting BSDs with the QNX hypervisor, and virtualizing BSD using the QNX Hypervisor. #### Guesting BSDs with the QNX Hypervisor The presentation provides insight into guesting BSDs with the QNX Hypervisor, including the virtualization environment, goals for the exercise, stories from the trenches, and an overview of the QNX host environment. BlackBerry holds the copyright for 20182019. All rights are reserved. The QNX Hypervisor design consists of multiple components: Guest OS (QNX), Guest OS (Linux), Pass-through Drivers, Drivers, BSP, Emulated devices, Virtual devices, Shared drivers, and a Shared message handler. At the heart of the Hypervisor is the qvm process, which manages virtual devices and memory. This system relies on the QNX Neutrino microkernel and the libmod_qvm.a module. The design supports multiple hardware configurations, including Aarch64 and x86-64 systems. BlackBerry's Hypervisor targets both QNX and Linux guests, with a focus on minimalism to reduce virtual firmware requirements. For instance, no BIOS emulation is used on x86_64. In terms of guest OS booting, Multiboot is utilized for QNX on x86, while Linux/x86_64 employs its own protected-mode loading protocol. Time management within guests can be complex. The Hypervisor operates as a process in the host system, scheduling virtual CPUs like normal threads. A privilege elevation interface allows guest code execution. A few design choices are notable: minimal environment, no or minimal virtual firmware, and the use of protected-mode loading protocols for Linux/x86_64.

Arm architecture reference manual armv8 r. Armv7-m architecture reference manual. Armv8 architecture reference manual pdf. Arm architecture reference manual. What is arm64 architecture. Armv8r reference manual. Architecture reference manual supplement armv8 for the armv8 r aarch32 architecture profile. Arm architecture reference manual supplement armv8 for armv8 r architecture profile. What is arm architecture.