A Novel SQLite-Based Bare PC Email Server

Hamdan Alabsi[1], Ramesh Karne[2], Alex Wijesinha[2(✉)], Rasha Almajed[2], Bharat Rawal[3], and Faris Almansour[2]

[1] College of Business, Mathematics and Science, Bemidji State University, Bemidji, MN 56601, USA
[2] Department of Computer and Information Sciences, Towson University, Towson, MD 21212, USA
awijesinha@towson.edu
[3] IST Department, Penn State University, Abington, PA 19001, USA

Abstract. We describe a SQLite-based mail server that runs on a bare PC with no operating system. The mail server application is integrated with a *server-based* adaptation of the popular SQLite client database engine. The SQLite database is used for storing mail messages, and mail clients can send/receive email and share files using any Web browser as in a conventional system. The unique features of the bare PC SQLite-based email server include (1) no OS vulnerabilities; (2) the inability for attackers to run any other software including scripts; (3) no support for dynamic linking and execution of external code; (4) a small code footprint making it easy to analyze the code for security flaws; and (5) performance benefits due to eliminating OS overhead. We describe system design and implementation, and give details of the bare machine mail server application. This work serves as a foundation to build future bare machine servers with integrated databases that can support Internet-based collaboration in high-security environments.

Keywords: Bare machine computing · Bare PC · SQLite · Database engine · Web server · Email

1 Introduction

Private email servers are often used for collaboration and communication among small groups that do not want to use popular public email systems such as gmail that rely on external servers, or within an organization, where a select group of individuals do not want to use the organization's own email system for privacy reasons. Because of the importance of protecting sensitive content, private email servers require stronger security than ordinary email servers. However, private email servers that require support of an operating system (OS) are harder to secure since they are vulnerable to exploits that target the OS, and have a larger attack surface even if a lean kernel or customized OS is used (since the

S. Kozielski et al. (Eds.): BDAS 2019, CCIS 1018, pp. 341–353, 2019.
https://doi.org/10.1007/978-3-030-19093-4_25

server requires external code to support the email application). A private email server that runs as a bare machine computing (BMC) application with no OS, kernel or external code provides an alternative to using a conventional private email server.

As a first step towards building a complete private email BMC system, we implemented a bare machine mail server. The current prototype uses a novel *server-side* adaptation of the SQLite client database engine to support mail system operations. SQLite [18] is widely used as a client-side database in many applications, smartphones, operating systems, and web browsers. SQLite is also used as a database engine for web sites and as a database for application servers. In all such cases however, SQLite requires the support of a conventional operating system (OS) or some form of a kernel. BMC SQLite has no OS dependencies and it would be possible to replace BMC SQLite itself with a BMC database engine customized for private email. The current SQLite-based mail server is accessed via the Internet. It can also be used to build a private email system that runs on dedicated network with no external connections.

The BMC email server only requires a bare PC, which is an ordinary desktop or laptop, with no other software loaded except for the application itself. The server application integrates lean versions of the necessary network protocol code with the application code. The application also directly communicates to the hardware and the network interface controller (NIC). BMC applications are intrinsically secure due to the absence of features that attackers typically target. A BMC system could thus be viewed as a form of minimalist system, where the tradeoffs are less features versus reduced avenues for attack. The remainder of this paper is organized as follows. In Sect. 2, we discuss BMC systems, and in Sect. 3, we present related work. Sections 4 and 5 respectively provide design and implementation details of the bare mail server. Section 6 contains the conclusion.

2 BMC Systems

The bare machine computing (BMC) paradigm, previously called dispersed operating system computing (DOSC) [11], enables applications to run on a bare PC with no OS or kernel. Details of the BMC application development methodology, which differs from that in a conventional system, are given in [12]. The BMC programming paradigm may be viewed as a holistic approach, where one needs to consider all aspects of application and system programming including the execution environment. As there is no OS or centralized kernel running in the system only a user mode is supported, and the programmer writes BMC applications that consist of a sequence of events executing in an interleaved fashion. Each event code becomes a small thread of execution that runs without any interrupts or task switching. Since there is no OS, the BMC programmer has total control of both application and execution flow. For network communication, BMC applications have their own Ethernet driver and a lean TCP/IP stack that implements only those protocols needed by the application.

In addition to eliminating the overhead due to the OS, BMC applications do not have OS vulnerabilities. They present fewer attack vectors than conventional systems since each application only implements a minimal set of necessary features. Being smaller and simpler, the BMC code is also easier to analyze for security flaws. Furthermore, since the bare PC server application is statically compiled and runs as a single thread of execution, it is difficult to compromise. Furthermore, no exploit code can run on the server. While dynamic linking and script execution can be disabled on a customized minimalist OS-based server, it still has more avenues for attack than a BMC server application. These characteristics of BMC systems make the bare mail server especially suited for private communication in high security environments. If needed, the bare PC file system located on detachable physically secured USB flash drives can be used by the server for permanent storage.

3 Related Work

SQLite on OS-based systems provides database services for websites and application servers [3]. Email servers such as [4,7,10] store emails in a SQL database. Code for a Linux/free BSD-based mail server with account storage in a backend database is given in [10]. A typical LAMP stack [13] consists of the Linux OS, Apache web server, MySQL database, and PHP, which could be used to build a customized mail server. NIST publications for securing servers include recent guidelines for secure hypervisor deployment [17]. Web, mail and database servers are frequently located in the cloud and virtualization is common. Conventional email servers have many external components, each of which would need to be secured in order to build a private email server. In contrast, the bare mail server runs without an OS or kernel and would have fewer components to secure. Alternative approaches to minimize OS impact include exokernel [5], bareMetal-OS [8], Linux kernel tinification [14], and the minimal Rust kernel [1]. Novel OSs, some of which are relatively small, are discussed in [16].

4 Design

4.1 Overview

Clients currently access mail and database services hosted on the BMC server using any Web browser (in future, BMC clients could be used for more security). The database client and database mail client data are stored on a USB flash drive. SQLite's command line interfaces are used for system administration and debugging. When the bare mail server is booted, any required files are transferred from a Windows system (using a lean version of trivial FTP), stored in memory, and used during server operation. If there is a pre-existing database, it will also be loaded. After the files are transferred, a dummy DB is created since SQLite needs a DB pointer to do any operation. A "mail" table defines the bare mail

server's database. User profiles with username and password are also loaded before the bare mail server application starts.

The integrated bare mail server application is statically compiled and currently runs on any Intel x86 architecture based PC or laptop. When the application executes, it is not possible to run scripts, add new applications, or load dynamic libraries. The bare mail server only performs functions that are defined at development time. New profiles can be loaded to add or delete users as needed.

Figure 1 shows the system architecture for the BMC server. Server components include the BMC Web server [9], an integrated SQLite USB file system [20], and a novel task system. The tasks and their interactions are shown on the left in Fig. 2. When the system boots up from the USB, the user menu for selecting services is shown. The "Load" option loads the integrated server application, and then the "Run" option runs the application, which starts the Main Task (MT).

The server has an MT, which creates, initializes, and stores other tasks in their respective task pools, and a Receive Task (RT), which receives network packets. The four task pools are USB Task (UT), SQLite Task (ST), HTTP Task (HT) and Post Task (PT), which reside in their respective stacks. When needed, a task is popped from the stack, placed into a circular list for execution, and pushed back onto the stack upon completion. The current tasks in a circular list are processed in a first-in-first-out (FIFO) manner. When a packet arrives, the MT starts the RT, which runs as a single thread of execution without interruption until the packet is processed. If the packet has a GET request, an HT is started, and if it has a POST request, a PT is started. A USB event such as insert, remove, read, or write will invoke a USB task to process the event. Similarly, a client's new SQLite event will start a SQLite task. For simplicity, we only consider one SQLite event (i.e., a single SQLite client), although the number of SQLite events (clients) is unlimited in principle.

4.2 SQLite Server

The BMC web server runs after booting. The display then shows the SQLite prompt and other information that are used for diagnostics and debugging. An administrator can also enter command line queries and get a response. Loading user profiles and creating a dummy database is done before clients access the database. Reading the "t1.sql" file makes the database ready for client operations, and the DBR flag is set in memory so that POST commands will be processed when the client makes requests. The server screen now shows that ST and MT are running. The server runs until it is powered down. Wireshark traces to validate network packets were obtained using an HTTP GET request for the "barerocks530k.gif" file, and for read mail where POST has the user name and GET is used to obtain the results. The SQLite code integrated in this system is taken from the amalgamated package [19], which has two "C" files: shell.c and sqlite3.c. The file shell.c has "main()" and other user interfaces.

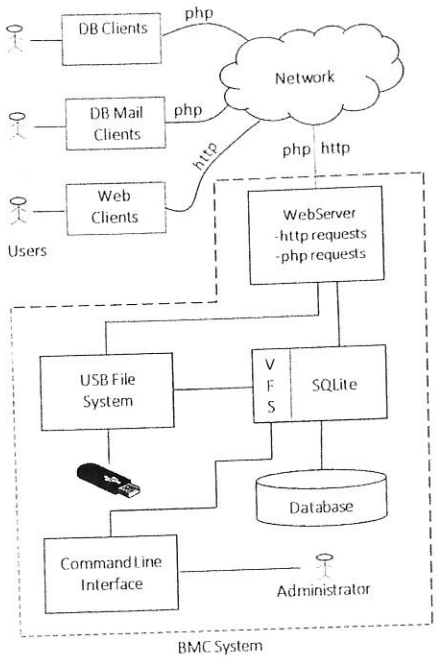

Fig. 1. System architecture

When this code was transformed to run on a bare PC, all system call interfaces were replaced with bare PC interfaces [15]. The right side of Fig. 2 shows the processing logic to integrate the bare PC SQLite system and tasks with the rest of the application. We changed the "main()" call in the shell.c file to "main_call()", so that there is only one main() in the system. The MT starts ST, which in turn calls "main_call()". The ST runs along with the other tasks in the integrated web server application. When ST starts, it prompts ">sqlite" and waits for the user requests. At the server side, all SQLite commands are run using this command line interface. The original SQLite is designed as a client. To build the bare mail server, we integrated SQLite with the bare PC web server by creating a dummy database. In "shell.c" there is an instance of a database, and appropriate initializations are done in the file "sqlite3.c". Otherwise, SQLite will not create any database instance. This approach allows us to use the SQLite code as is and simply tap into the database interface in the file "shell.c". A small script file t1.sql is used for this purpose. It includes the do-meta-command "read t1.sql" to create a dummy database and a dummy table t1. We can enter other SQL statements or meta-commands at the command prompt before running the script file. Once the script file is run, the SQLite interface at the command line interface is disabled as the ".DBR" meta-command sets a DBR flag in shared memory. The ".DBR" meta-command is added to the SQLite set of commands.

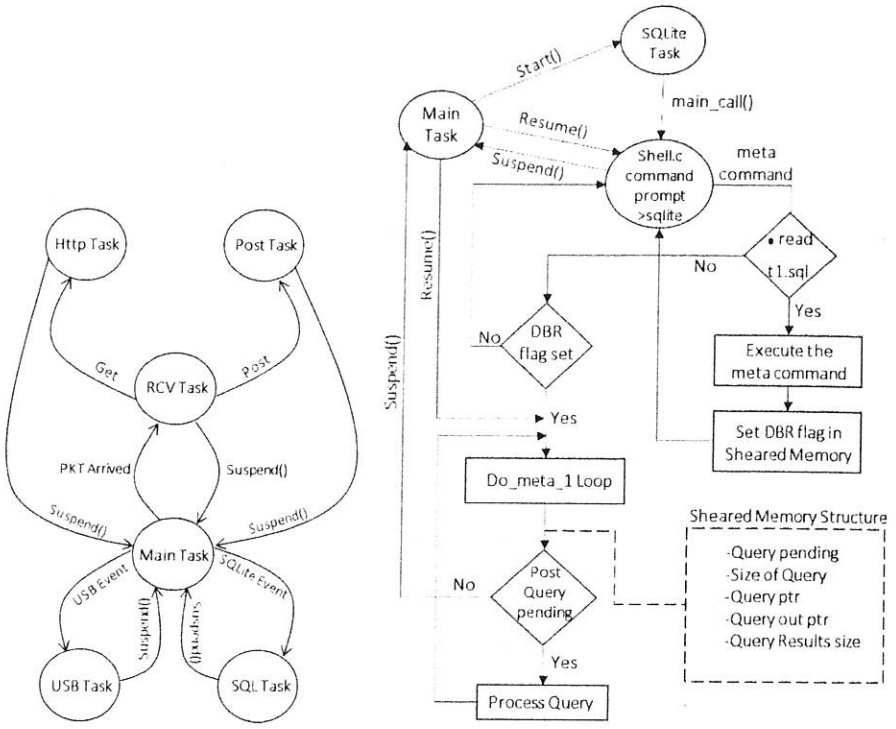

Fig. 2. Task interface (left) and task-based SQLite query processing (right)

When the DBR flag is set, the SQLite shell program will go into the "do_meta_1" loop. When necessary, ST suspends itself and waits for client requests. The suspend() function returns to the MT. The suspend() function in C++ was mapped to the suspend() function in C (as class instantiation cannot be done from our C program). Once the DBR flag is set, SQLite processes client queries via a POST command.

4.3 Mail Server Operations

A network event in the server application occurs when a TCP SYN packet arrives. Figure 3 shows control flow in the server including mail operations and the relevant TCP states and packets. Either an HTTP GET request or a HTTP POST request is sent by the client. When the client needs data from the server a GET request is used, and when it sends data to the server a POST request is used. For a GET request, the server knows how many packets to send and it can control the data transfer. For a POST request, the client controls the data transfer. In this case, the server keeps track of the number of packets and the total data size to verify that all packets have arrived. POST data processing requires a PHP parser. To avoid complexity, only the functionality needed

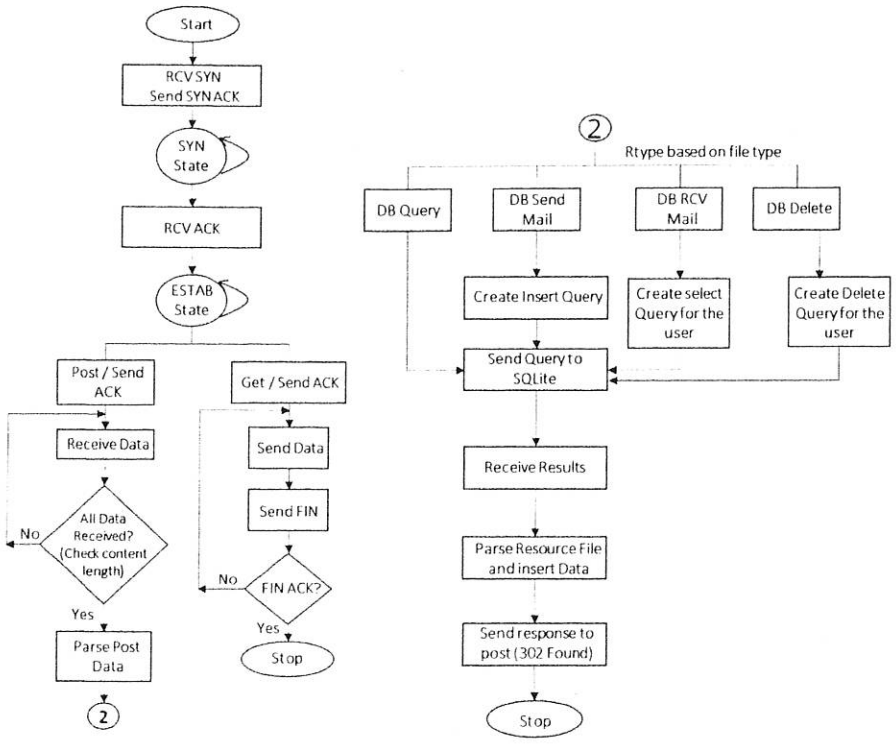

Fig. 3. TCP flow and mail operations

for the integrated application is implemented in the PHP parser. Since we are designing a server system, it is possible to control the design of the PHP files and their contents.

It should be noted that HTML files are static content and PHP files are dynamic content. When the client accesses the database, dynamic content is required due to the nature of queries and mail requests. POST data is received a packet of data at a time until all packets are received. Then the server processes the data according to the type of the request indicated by RType (circled value 2) in Fig. 3. There are five types of client-server interfaces, which result in five forms. Figure 4 shows the relation between functions, file names and operations (Rtype for login is not shown in Fig. 3). Each form corresponds to a different type of PHP file. In addition, the client fetches the results from the database using a separate form named "inbox.php". Thus a total of six PHP files are required in the system. Parsed attributes of each file are also shown in Fig. 4.

When the "login.php" file is received as POST data, the server parses the data for the username and password, validates them, and logs the information. This information is used throughout the session for a given user when logged in. Only necessary information is parsed using keywords; the file also has keywords to help the parser find the key reference points. Database queries from a client are

Function	File name	Operation
Login	login.php	Extract username, password
DB Query	Compose.php Inbox.php	Extract query Put results
Send Mail	Sendmail.php	Extract mail parameters Form insert query
Receive mail	Retrieve.php Inbox.php	Extract username Form select query Insert results in file
Delete mail	Delete.php	Extract username Form Delete query

Fig. 4. Functions, file names and operations

received at the server via a "compose.php" file sent using the POST command. The server extracts the query data from the packet(s) and forms a query block for the SQLite server, which returns the results in a memory block. These results are inserted into the "inbox.php" file at the insertion points shown in Fig. 5. The insertion points "POINT1" and "POINT2" provide the references in the files to insert data into the file. When the result data is inserted, the "POINT2" reference will be moved down to increase the file size according the size of the data. This technique enables us to avoid writing a full PHP parser.

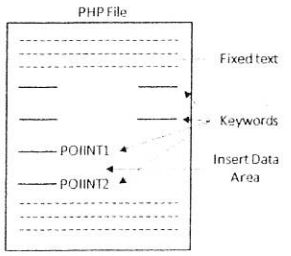

Fig. 5. Inserting results in the PHP file

When sending mail, the "sendmail.php" file is used to form a mail message. At the server, mail attributes such as from, to, subject, and body are extracted, and an INSERT query is formed. This query is sent to SQLite for execution and checked for a valid return code. Parsing of the "sendmail.php" file is similar to the other PHP files. In this case, a username is also parsed from the mail content and validated by comparing with the login name. Receiving mail is slightly more complex than other client requests. The "retrieve.php" file sent to the server via POST contains the username of the receiver. A SELECT query is formed to retrieve emails for this receiver from all other users (if any). This query is sent to SQLite and the results are inserted into the "inbox.php" file. The client sends a GET request to obtain the results. Inserting results into the file "inbox.php" is similar to inserting database query results. Finally, the "delete.php" file is received by the server as POST data. The server extracts the username from the message and the appropriate query is formed to delete the mail message from the database. For a given user, mail from a sender is deleted based on the sender's username. Only one mail message at a time can be deleted.

The server is a prototype that integrates client SQLite as a server component, and handles basic database queries and mail requests. While more features can be added to the server as it is designed in a modular fashion, this may increase server complexity and result in reduced security.

5 Implementation

SQLite integration is implemented in C/C++ with direct hardware interfaces. Communicating from C code to C++ code required the use of special pointer conversions and prototypes. The server application uses an Intel Gigabit Ethernet NIC (on-board chip) and its corresponding BMC driver. The executable size is 821,248 bytes. This includes the application code and the required execution environment code that makes it a self-contained, self-managed and self-controlled application. A user owned USB drive contains this executable including its boot code, which is used to boot up a bare PC and run the server application. The application runs on any Intel x86 architecture based PC or laptop. The SQLite and POST source code are 140K lines and 3K lines.

The SQLite DB application runs as a client server system. When a user clicks on the DB link to access the database or DB mail, the login interface appears. The user enters the username and password (which are validated), and the system menu is displayed as in Fig. 6. The page for composing a mail message is shown in Fig. 7. The results of a query are shown in Fig. 8. Selecting the DB Query option, a user can enter a SQLite query as in Fig. 9. Database queries can be typed in as text in the window or they can be included in an attachment. The system is tested for both options. A database application running on the system can be accessed by users and administrators. The bare PC web server has a debug mode, where some web pages can show internal server operations and database operations. Database queries are limited to the queries that can be performed by SQLite.

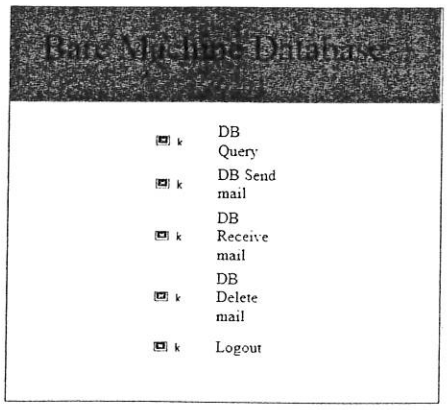

Fig. 6. Menu screen

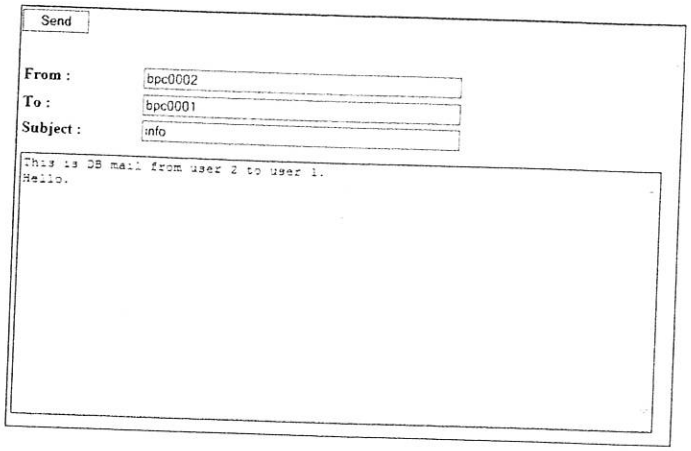

Fig. 7. Compose mail page

The bare mail server is designed to work with a limited group of users who share the SQLite database to send and receive email after logging in to the server. The server does not include TLS at present, but it can be added by integrating the existing BMC TLS code in [2]. An SMTP-based bare PC email server built previously [6] could also be integrated with the present system if needed to communicate with external mail servers. The SQLite database and user accounts are created by an administrator in a physically secure manner. An OS-based web browser is used to send, receive/read, and delete email messages. The corresponding database operations are send, receive, and delete. Sent messages are stored in the database. The send mail form requires the sender's name

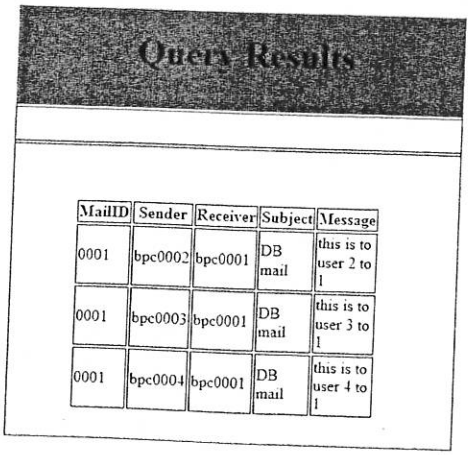

Fig. 8. Query results page

address, receiver's name, subject, and message body. This form is sent to the server as a POST message. The server inserts this query into the mail database and returns. The receive mail form requires the username for the receiver. A page similar to that shown in Fig. 8 is used to display the results of receive mail. Currently, all emails received for a user are displayed. The server receives the username and password and forms a query. It submits the query to SQLite and collects the results, which are stored in a file and returned to the client via a GET request. The delete mail form requires the sender's username for the message to be deleted. A user must be logged in (as a receiver) to delete a sender's messages.

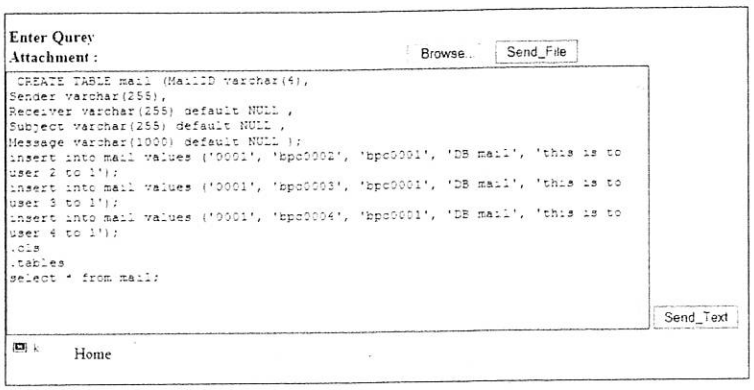

Fig. 9. Query entry page

6 Conclusion

We built a novel email server that provides mail services on a bare PC by running SQLite in a client-server environment. The server enables a small group of users to collaborate by sharing files or communicate privately without using a public email system. It is immune from attacks that require an underlying OS and is easier to secure than a conventional email server due to its reduced attack surface and smaller footprint. The code is currently written based on the Intel IA-32 CPU architecture, but can run in principle on other CPU architectures by developing the necessary hardware interfaces. The email server extends a bare PC webmail server built earlier in three significant ways: (1) it has the capability to store messages on USB-based external mass storage; (2) it uses the bare SQLite database engine and associated queries for message storage and retrieval: (3) it is integrated with a USB-based bare PC file system. The server is accessed by using any conventional OS-based Web browser. Since the browser/mail client can be compromised due to OS vulnerabilities, only the mail server retains the advantages of being resistant to OS-based attacks. For better security and enhanced privacy, the web browser should also be run on a BMC

system. Such a web browser does not exist at present, but a text-based bare PC web browser is currently under development. While hardened lean OS-based private email servers can also be run on dedicated networks and limited to a select group of users, they are still vulnerable to OS-based attacks. This work serves as a foundation to build secure private bare PC email servers and clients that run with no OS support.

References

1. A Minimal Rust Kernel. https://os.phil-opp.com/minimal-rust-kernel/. Accessed 31 Jan 2019
2. Appiah-Kubi, P., Karne, R.K., Wijesinha, A.L.: A bare PC TLS webmail server. In: 2012 International Conference on Computing, Networking and Communications (ICNC), pp. 149–153. IEEE (2012)
3. Appropriate Uses for SQLite. https://www.sqlite.org/whentouse.html. Accessed 31 Jan 2019
4. Dbare mail: fast and scalable SQL based email services. http://www.dbaremail.org/. Accessed 31 Jan 2019
5. Engler, D.R., Kaashoek, M.F., et al.: Exokernel: an operating system architecture for application-level resource management, vol. 29. ACM (1995)
6. Ford Jr., G.H., Karne, R.K., Wijesinha, A.L., Appiah-Kubi, P.: The design and implementation of a bare PC email server. In: 2009 33rd Annual IEEE International Computer Software and Applications Conference, vol. 1, pp. 480–485. IEEE (2009)
7. Git-Hub – nodemailer/wildduck. https://github.com/nodemailer/wildduck. Accessed 31 Jan 2019
8. GitHub – ReturnInfinity/bareMetal-OS. https://github.com/ReturnInfinity/bareMetal-OS. Accessed 31 Jan 2019
9. He, L., Karne, R.K., Wijesinha, A.L.: The design and performance of a bare PC web server. Int. J. Comput. Appl. **15**(2), 100–112 (2008)
10. iRedMail-Freem Open Source Mail Server Solution. https://www.iredmail.org/. Accessed 31 Jan 2019
11. Karne, R.K., Jaganathan, K.V., Rosa Jr., N., Ahmed, T.: DOSC: dispersed operating system computing. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 55–62. ACM (2005)
12. Khaksari, G.H., Karne, R.K., Wijesinha, A.L.: A bare machine application development methodology. Int. J. Comput. Appl. **19**(1), 10–25 (2012)
13. LAMP Stack. https://www.turnkeylinux.org/lampstack. Accessed 31 Jan 2019
14. Linux Kernel Tinification. https://tiny.wiki.kernel.org/. Accessed 31 Jan 2019
15. Okafor, U., Karne, R.K., Wijesinha, A.L., Rawal, B.S.: Transforming SQLite to run on a bare PC. In: 7th International Conference on Software Paradigm Trends (ICSOFT), pp. 311–314 (2012)
16. Operating Systems You May Not Have Heard Of (But Should). https://www.hongkiat.com/blog/lesser-known-operating-systems/. Accessed 31 Jan 2019
17. Security Recommendations for Hypervisor Deployment on Servers, NIST Special Publication 800-125A, January 2018
18. SQLite. https://www.sqlite.org/index.html. Accessed 31 Jan 2019
19. The SQLite amalgamation. https://www.sqlite.org/amalgamation.html. Accessed 31 Jan 2019

20. Thompson, W., Karne, R., Wijesinha, A., Chang, H.: Interoperable SQLite for a bare PC. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2017. CCIS, vol. 716, pp. 177–188. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58274-0_15