BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATED TESTING OF SMART CARD AUTHENTICATION IN GUI

AUTOMATICKÉ TESTOVÁNÍ GUI PRO AUTENTIZACI POMOCÍ ČIPOVÝCH KARET

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

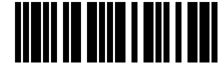
AUTHOR — **ONDŘEJ MACH**
AUTOR PRÁCE

SUPERVISOR — **Dr. Ing. PETR PERINGER**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment



148710

Institut: Department of Intelligent Systems (UITS)
Student: **Mach Ondřej**
Programme: Information Technology
Specialization: Information Technology
Title: **Automated Testing of Smart Card Authentication in GUI**
Category: Software analysis and testing
Academic year: 2022/23

Assignment:

1. Get familiar with principles and tools for automated software testing with focus on testing of GUI in Linux. Learn about tooling and infrastructure currently used for testing of smart cards in Red Hat.
2. Find the most appropriate strategy for testing of user authentication in GUI. Design the suitable way to integrate automated testing of GUI for smart card authentication into the current Red Hat testing infrastructure.
3. Implement the solution based on the design (i.e. implement the module of SCAutolib providing new functionality for GUI testing). Prepare a set of simple test cases to demonstrate real-life contribution of this work.
4. Evaluate the results and propose possible future development directions.

Literature:

- YADLOUSKI, Pavel. Automated Testing of Smart Cards. Brno, 2021. Bachelor’s thesis. Brno University of Technology, FIT.

Requirements for the semestral defence:

- First two points of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Peringer Petr, Dr. Ing.**
Consultant: Mgr. Marek Havrila PhD.
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: 1.11.2022
Submission deadline: 10.5.2023
Approval date: 3.11.2022

Abstract
The aim of this thesis is to automate testing of the most common use cases of smart card authentication in Red Hat Enterprise Linux (RHEL). These include logging in with GDM, using lock-on-removal feature in GNOME shell and unlocking the system. Because these use cases include interaction with the desktop manager, conventional testing tools cannot be used. A new module was added to an existing Python library to capture the screen and control the system under test. The implementation does not depend on a specific display server. A set of common test cases, which had been previously tested manually, was implemented. The solution will be used in Red Hat to test new releases of RHEL.

Abstrakt
Cílem této práce je automatizovat testování běžných případů autentizace pomocí čipových karet v systému Red Hat Enterprise Linux (RHEL). Mezi ně patří přihlašování v GDM, použití funkce lock-on-removal v prostředí GNOME shell a odemykání systému. Protože tyto případy použití zahrnují interakci s přihlašovací obrazovkou, nelze použít běžné testovací nástroje. Výsledkem práce je modul v jazyce Python, který umožňuje snímat obrazovku a ovládat testovaný systém. Dále byla implementována sada běžných případů užití, které byly dříve testovány manuálně. Řešení bude použito ve firmě Red Hat k testování nových verzí systému RHEL.

Keywords
Automated GUI testing, Linux, smart cards, authentication, GDM, OCR, Python

Klíčová slova
Automatické testování GUI, Linux, čipové karty, autentizace, GDM, OCR, Python

Reference
MACH, Ondřej. *Automated Testing of Smart Card Authentication in GUI*. Brno, 2023. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Dr. Ing. Petr Peringer

Automated Testing of Smart Card Authentication in GUI

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Dr. Ing. Petr Peringer. The supplementary information was provided by Mgr. Marek Havrila PhD., my external consultant from Red Hat. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Ondřej Mach
May 5, 2023

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Ing. Petr Peringer for his invaluable guidance and support. I would also like thank Mgr. Marek Havrila PhD., who has provided me with meaningful advice and encouragement that enabled me to complete the project.

Contents

1 Introduction **3**

2 Background **5**
2.1 Software testing overview . . . 5
2.2 Automated GUI testing . . . 6
2.3 Smart cards . . . 7
2.4 Smart card authentication in RHEL . . . 8
2.5 Smart card testing in Red Hat . . . 10
2.6 Windowing systems in Linux . . . 13

3 Existing solutions **17**
3.1 Python with Accessibility API . . . 17
3.2 openQA . . . 18
3.3 Visual GUI testing tool with VNC . . . 18

4 Design of a new solution **21**
4.1 Requirements . . . 21
4.2 Accessing the framebuffer . . . 22
4.3 Methods of screen contents recognition . . . 23
4.4 Emulation of user input . . . 24
4.5 Integration into SCAutolib . . . 24

5 Implementation **26**
5.1 Environment used for development . . . 26
5.2 Screen capture using ffmpeg . . . 26
5.3 Screen contents recognition . . . 27
5.4 Keyboard . . . 30
5.5 Mouse . . . 30
5.6 Test setup . . . 32
5.7 Reading system logs . . . 33
5.8 GUI module's API . . . 34
5.9 Test cases . . . 35
5.10 Running the tests in Red Hat infrastructure . . . 37

6 Evaluation of implemented solution **38**
6.1 Screen capture and OCR . . . 38
6.2 Running the tests in OpenStack . . . 38

7 Conclusion **40**

Bibliography **41**

A Individual test cases with description **45**

B Files for running the test in Red Hat infracructure **50**

C Log from successful test run **54**

Chapter 1

Introduction

Software testing has become a standard for any kind of software project. Red Hat Enterprise Linux (RHEL) [33] is no exception in this aspect. Every component is thoroughly tested before it is shipped to the customer. For complex software products with large amount of tested components, test automation is especially important, as it provides traceable and reproducible results. Automated tests can run faster and more often, reducing needless bugs and leading to faster delivery of the product. Many tests for RHEL are already automated, but smart card authentication in GUI (Graphical User Interface) is still tested manually. The aim of this thesis is to automate the tests for GUI authentication using smart cards for RHEL.

The chapter 2 covers important concepts and terminology, providing a wider background for this work. It explains concepts from software testing, and presents the smart card technology. Additionally, the chapter outlines how smart card functionality is tested in Red Hat and introduces *SCAutolib* – a library designed specifically for automated testing of smart cards. *SCAutolib* was developed as a Bachelor's thesis [49], and current work represents a continuation of that effort. The background chapter also explains the windowing systems [29] that are used on Linux, since their understanding is essential for design of the new GUI testing tool.

In chapter 3, available GUI testing tools are explored. By using an existing tool, we can avoid the significant time investment needed to develop a new one. The biggest hurdle we have faced in this undertaking is testing the display manager. Display manager is the GUI application that allows users to log into the system. After the user authenticates successfully, the display manager starts a session for the user. Unfortunately, we did not find any GUI testing tools that were suitable for this project, meaning that a new solution had to be created.

The chapter 4 focuses on designing a new solution that would solve the problems of existing ones. First, we have specified the requirements for the new solution. The chapter goes through individual problems and chooses the technologies that best satisfy the requirements. It discusses the decision to use OCR (Optical Character Recognition) [1]. OCR is a process that converts an image of text into machine-readable text format. In case of GUI testing frameworks, OCR is used to locate GUI elements that are labeled with text.

The chapter 5 delves into the implementation details of the project. Most importantly, it discusses the screen capture and input device emulation in system under test (SUT). It also describes the method of processing the screenshots and finding the locations of GUI elements.

The chapter 6 analyzes the performance of the newly implemented solution. We have assessed the performance of the implementation in a virtual machine, as well as test execution time on the Red Hat Infrastructure. Although test execution time is not a significant factor, we have ensured that the time spent on screen capture and image processing in not excessive.

Chapter 2

Background

This chapter covers a range of topics related to the thesis. First, it describes various approaches for automated testing of graphical user interfaces. The chapter also explains the basics of smart cards and how they are used in RHEL. It then illustrates, how this functionality is tested in Red Hat. Last, the chapter provides an overview of windowing systems that are used in Linux.

2.1 Software testing overview

Software testing [12] is an important part of software development, which ensures that the software behaves as expected. There are different approaches to software testing, each one serving a specific purpose. This section discusses various types of software testing, which are relevant to this work.

2.1.1 Static and dynamic testing

Software testing can be classified into two categories: dynamic testing and static testing. While dynamic testing involves running the software and testing its behavior in different scenarios, static testing focuses on examining the software's source code, design, and documentation without running the actual software.

Dynamic testing, also known as functional testing, focuses on testing the software's functionality. This technique encompasses creating test cases, which cover various scenarios, and executing them. The dynamic testing verifies that the software responds correctly to the given input. On the other hand, static testing does not involve running the software. Static testing is mostly performed manually using techniques such as code review, inspection, and walk-through. The objective of static testing is to identify defects before the software is executed. Although both of the approaches are important to ensure the software's quality, this work focuses entirely on the dynamic testing.

2.1.2 Black and white box testing

Another way to classify software testing is black and white box testing. Black box testing focuses on testing the software without considering its internal structure or code. The tests are not concerned with how the software works, but rather with how it responds to different inputs. White box testing, on the other hand, takes the software's internal structure into account. The test cases of white box testing should reflect the structure of the tested code.

White box testing also makes use of code coverage, which can measure the percentage of the code covered according to some criterion.

2.1.3 Integration testing

Integration testing tests the software system as a whole. Typically, software systems comprise of multiple components, which interact with each other. The integration testing is not concerned with the interaction between the components. Instead, it tests whether the whole system behaves as expected.

2.1.4 Sanity testing

Sanity testing focuses only on the most critical functionality of the software. The purpose of sanity testing is to verify a new build of software as quickly as possible. If the new build is stable enough, more extensive testing can be started. If any issues are identified during sanity testing, the testing process will stop, and the software development team will need to fix the defect before proceeding.

2.1.5 Regression testing

Regression testing is performed after changes have been made to the software. The objective of regression testing is to ensure that the modifications have not introduced any new defects. Regression testing involves executing the existing test cases to verify that the software or system still works. Compared to sanity testing, the regression testing can be time-consuming, but it is much more comprehensive.

2.2 Automated GUI testing

Graphical user interfaces (or GUIs for short) have been widely available since early 1980s. At first, they were tested manually, but the field of automated GUI testing has emerged soon after. The automated testing started in 1990s and the approaches to testing them have been developing alongside the GUIs.

Since the time they were introduced, the GUIs have developed a great deal and so has the automated testing. Most of the approaches to graphical testing can be divided into 3 main categories [10]. These categories are listed below in chronological order.

- 1$^{\text{st}}$ generation tools: The 1$^{\text{st}}$ generation tools use fixed coordinates to locate graphical elements on the screen. The coordinates of elements are hard-coded by the tester. This approach worked well with systems, where the application is always displayed on full screen and cannot be resized. With the arrival of newer operating systems, where the windows can be resized, this approach became obsolete. This raised the need for a new generation of testing tools.

- 2$^{\text{nd}}$ generation tools: The 2$^{\text{nd}}$ generation tools are most commonly used approach today. They are able to access the information about graphical elements via API. Thus they can read element's position dynamically and solve the problem of resizing windows. *Selenium* [3] is an example of a 2$^{\text{nd}}$ generation tool for testing web apps. It can find the element in the website structure using functions like `find_element_by_name` and `find_element_by_id`. A limitation of these tools is that one tool cannot test

programs created in every GUI framework. This is because this approach relies on a specific API to find the elements, but every GUI framework uses a different one.

- 3rd generation tools: The 3rd generation tools, known as Visual GUI testing (VGT) tools, are able to detect graphical elements from an image. They do not rely on any API to get the element positions, the only thing needed is the contents of the screen. In this aspect, the testing tools very much like a manual tester. Compared to the 2nd generation tools, these can be truly universal in the sense that they can test any application. In order to detect the graphical elements, these tools have to perform image matching or optical character recognition. For this reason they are usually slower than 2nd generation tools. Tests written using these tools tend to require more maintenance, because every change in the style can stop the recognition from working. Examples of 3rd generation tools are *SikuliX* [13] (open source) or *EyeAutomate* (commercial).

2.3 Smart cards

Smart cards [18], often called chip cards or integrated circuit cards, are electronic devices used for authentication purposes. They are usually constructed from plastic and have the same size as a credit card (see figure 2.1). While certain cards rely on metal contacts connected to the chip for communication, others are contactless, and some cards support both communication interfaces. The most common applications of smart cards include credit cards, SIM cards or public transport cards. Some national identity cards are equipped with a chip as well. It should be noted that not all smart cards have equal hardware capabilities. This work mostly focuses on cards with a secure cryptoprocessor, more on which can be found in section 2.3.3.

Figure 2.1: USB smart card reader with one smart card inserted and a second one placed in front of it

2.3.1 Physical characteristics

No matter the hardware capabilities, the vast majority of smart cards conform to ISO/IEC 7810 standard [16]. This standard defines multiple card sizes, namely ID-000, ID-0, ID-2, ID-3. The ID-1 format is the most common one, it is the format of credit cards. Cards of all sizes should have their thickness ranging from 0.68 to 0.84 millimeters The standard also specifies other properties like bending stiffness or resistance to heat and humidity.

It should be noted that ISO/IEC 7810 does not specify the technology used in the cards. Instead of the integrated circuit, the card may use a magnetic stripe to carry information.

2.3.2 Contact cards

The contact card interface is defined by ISO/IEC 7816 [15]. The standard describes the locations of the contacts and their electrical characteristics. It also defines a transmission protocol between the card and the reader.

The standard also includes part 7816-15: Cryptographic information application. The newest revision from 2016 defines the format of cryptographic information and the sharing mechanisms. Specifically, the part defines storage and retrieval of cryptographic information on the card. It also describes different authentication mechanisms and cryptographic algorithms.

2.3.3 Cryptographic smart cards

Not all smart cards [18, 27] have cryptographic capabilities. The least expensive smart cards, often called memory cards, contain only EEPROM chip. EEPROM (Electrically Erasable Programmable Read-Only Memory) is a type of non-volatile memory. The EEPROM chip inside memory cards communicates directly with the reader.

The more advanced cards are called microprocessor cards. These cards contain a microprocessor, RAM, ROM, and sometimes a cryptographic module. The processor can be 8-bit, 16-bit or a 32-bit and is executing code from ROM (Read-Only Memory). This code is a specialized operating system. The processor sits in between the reader and the EEPROM, so reading memory directly is no longer possible. This enhances the card's security, since the processor can deny reading some regions of memory.

Cryptographic smart cards [21] are devices that store and manage private keys and perform cryptographic operations. An example of such operation is creating a digital signature using a private key that is stored on the card. The main idea of cryptographic smart cards is that the cryptographic operations can be done internally, without the private key getting out of the card. If the system is compromised, the attacker cannot steal the private key. Additionally, these cards implement various tamper resistance features to prevent extracting the key.

2.4 Smart card authentication in RHEL

The software stack for smart cards in RHEL comprises many software components. It starts with USB drivers in the kernel and has multiple layers in user space (see figure 2.2). This section goes through some of the components involved in smart card authentication.

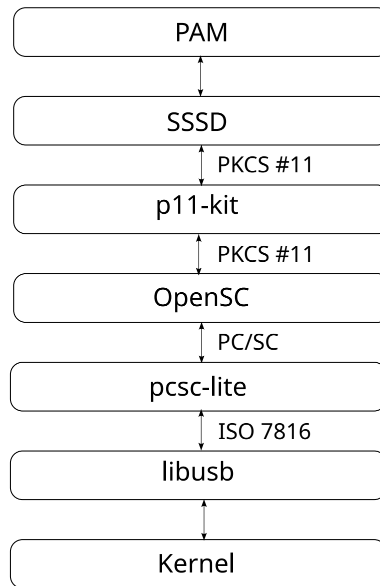


Figure 2.2: Software stack used for smart card authentication

2.4.1 pcsc-lite

pcsc-lite [37] is a piece of middleware, which communicates with the card reader over USB and provides PC/SC API [30]. PC/SC, standing for Personal Computer/Smart Card, is a standardized API for smart card readers. PC/SC is platform independent, implementations exist on all major operating systems. *pcsc-lite* is the most commonly used open source implementation. It is distributed under BSD licence and it is used in most Linux distributions and BSD operating systems. *pcsc-lite* is built on top of *libusb* which is used to communicate with the reader over USB.

2.4.2 OpenSC

OpenSC [26] is a set of libraries and tools for working with cryptographic smart cards. It implements PKCS #11 API and some other platform-specific APIs. OpenSC is multi-platform software and it is supported in Linux, BSD systems, Windows and MacOS. PKCS, standing for Public Key Cryptography Standards, is a group of standards defined by RSA Laboratories. PKCS #11 [11], being one of the standards, is the most commonly used API [25] for cryptographic smart cards. PKCS #11 is also known as Cryptoki, which stands for Cryptographic Token Interface.

2.4.3 p11-kit

p11-kit [9] is used to load and enumerate all PKCS #11 modules. It provides a consistent configuration setup for installing the modules in a discoverable manner. *p11-kit* is developed by Freedesktop.org and it can be built on any UNIX-like system.

2.4.4 SSSD

SSSD [39] is an acronym for System Security Services Daemon. It is a service that provides authentication and authorization for its client applications. *SSSD* supports various authentication mechanisms, including smart card authentication. Besides that, *SSSD* is often used to authenticate users against identity management system such as *FreeIPA* or *Active Directory*. *SSSD* communicates with the smart card via PKCS #11 API. For the client applications, *SSSD* provides multiple responders, including but not limited to *NSS*, *PAM*, and *SUDO*.

2.4.5 PAM

PAM [19] stands for Pluggable Authentication Modules. It is a framework used in UNIX-like systems, which provides a modular way to handle authentication and authorization in user applications. *PAM* works by intercepting requests from applications and forwarding them to the appropriate *PAM* module. The module then performs the necessary checks and returns a result to *PAM*, which either grants or denies access. Both GDM [42] (GNOME display manager) and GNOME shell use *PAM* for authentication.

PAM can be set up using configuration files, which are usually located in `/etc/pam.d/` directory. Although these files offer great flexibility, their complexity makes them prone to errors when edited manually. Red Hat based Linux distributions solve this issue with *authselect* [32] utility. *authselect* allows the user to configure authentication by selecting a specific profile. *authselect* also provides profiles for advanced authentication features such as smart card and fingerprint authentication. These features would be difficult to configure manually. For smart card authentication *SSSD* profile must be selected with appropriate options (see section 2.5.1).

2.5 Smart card testing in Red Hat

Since RHEL officially supports smart card authentication, Red Hat has to test this functionality. Currently, only some tests for smart card authentication are automated. These tests are exclusive to authentication in command line interface (CLI). Smart card authentication in GUI is still tested manually.

2.5.1 Manual GUI testing

Currently, smart card authentication in GUI is tested manually. This process consists of many steps and it is quite time-consuming. First, the tester downloads an image of tested RHEL version and sets up a virtual machine. Then he follows the steps for setting up the system. To set up smart card authentication, multiple configuration files need to be changed and a certificate has to be added to the machine. After everything is set up, the tester connects a card reader to the virtual machine via USB pass-through.

Currently, three main features are tested. One of them is logging in using GNOME Display Manager (GDM). This feature needs to be tested with different *SSSD* (System Security Services Daemon) parameters. For example, the system can be set up for optional or required smart card authentication. If the smart card is optional, the user is allowed to log in with his password. With the smart card required, the user cannot log in using password and is prompted to insert the card. All the test cases are also tested with incorrect password or PIN to verify that unauthorized access is not possible.

The other features to test are the GNOME lock screen and lock on removal. After the user is logged into GNOME session, he can enter lock screen from power menu or using a keyboard shortcut. The lock screen should take into consideration, whether the user has previously logged in using smart card or using his password. To unlock the session, the user should use the same authentication method that was used to start the session. The lock on removal feature can be enabled by another *SSSD* parameter. When the feature is enabled, the GNOME session should enter lock screen, as soon as the smart card is removed from the reader.

The tester verifies that all the features work as expected. After that, he writes a short report in the test case management system. Overall, the process of manually testing smart card functionality is rather labor-intensive, which could be improved with automation.

2.5.2 Infrastructure

When the tests are automated, they are usually deployed to run automatically on Red Hat's internal infrastructure. This way, the tests can be executed as soon as a new build of RHEL is ready for testing. The setup, which is described in the paragraphs below, is specific to Crypto quality engineering team.

The tests run mostly in virtual machines in cloud. There are multiple platforms, where the virtual machines can be allocated, including OpenStack or Amazon Web Services. Regardless of the platform, all the tests share metadata with common format. These include information about the test, or its expected execution time. They also specify RPM packages that should be installed before the test runs. Lastly, the metadata determine the entry point for the tests, which is usually a shell script. In order to deploy the tests into the infrastructure, at least the metadata and the script are required.

In case of simple tests, the actual test cases are written directly in the entry point script. However, the tests for smart cards are too complex to be written in single script. For testing smart cards, the script installs *SCAutolib* [48] and clones the tests from git repository. After that, it executes the test cases using pytest framework. More on this process can be found in the next section.

2.5.3 SCAutolib

Smart card testing requires an extensive setup and it interacts with many software components while the test is performed. Both of these issues are solved by *SCAutolib* [48, 49]. *SCAutolib* is a Python library developed by Red Hat specifically for smart card testing. The library has two main use cases. It is used to set up the system under test. When the test cases are run, library abstracts the system's resources to the test.

SCAutolib can be downloaded from Python Package Index (PyPI) or cloned directly from git repository. After the library is installed, it can be used to set up the system for smart card testing. For this purpose, library provides a command named `scauto`.

```
scauto --conf conf.json prepare --install-missing
```

The command above sets up the system under test according to `conf.json` file, which is supplied as a command line option. This configuration file specifies the users, their smart cards and certificate authorities for the cards. As the first step of the setup, the

library checks all the dependencies for running the tests. Those are mostly RPM[1] packages for smart card support. The `--install-missing` option makes the dependencies install automatically. If the option is not present, the library still checks the packages and reports the missing ones. When the setup runs in the Red Hat infrastructure, the packages should be already installed, since they are specified in the test metadata.

Next, *SCAutolib* creates the users for testing. Configuration file includes details about certificate authorities, users and smart cards. Users can authenticate locally or using a remote authentication server. In the file, each user has a password and a PIN for the smart card. After the users are created, the smart cards are set up. Each card must contain a certificate for the user. This certificate can be signed by a certificate authority. The configuration file allows selecting between local certificate authority or a remote certificate server[2].

Then, the certificate is written to the smart card. As of now, *SCAutolib* uses only virtual smart cards emulated using *virt_cacard*. *virt_cacard* emulates the virtual card on the level of ISO 7816. It connects over TCP socket to *pcsc-lite*, which has a special driver for emulated cards. This means that most of the components in the smart card authentication stack are tested (see section 2.4). There are plans to support testing with real smart cards; however the feature has not been implemented yet. The other purpose of *SCAutolib* is to provide an abstraction layer for the tests. This is best explained on an example of a simple test.

```
from SCAutolib.models.authselect import Authselect

def test_su_login_with_sc(local_user, user_shell):
    with Authselect(required=False), local_user.card(insert=True):
        cmd = f'su {local_user.username} -c "whoami"'
        user_shell.sendline(cmd)
        user_shell.expect_exact(f"PIN for {local_user.username}:")
        user_shell.sendline(local_user.pin)
        user_shell.expect_exact(local_user.username)
```

This example was taken from SC-tests git repository[3], which contains the tests that use *SCAutolib*. The function is run using pytest framework, which supplies the `local_user` and `user_shell` fixtures. `local_user` is an abstraction of a user provided by *SCAutolib*; `user_shell` is an abstraction of shell from *pexpect* library. When the test starts, authentication is configured using `Authselect` and the user's card is inserted. Both of these activities are done using abstractions from *SCAutolib*. The test script itself does not have to call `authselect` command directly. It is also independent of the card implementation, which could be either virtual or physical. This particular test runs command `su` to switch the user and `whoami` to print out the username. Then the script expects to be asked for authentication using PIN code for smart card. By virtue of abstraction, none of the values have to be hard-coded. At the end of the test, it is checked whether the `whoami` command ran successfully.

[1] RPM (RPM Package Manager) is a package management system used in Red Hat based Linux distributions.

[2] For purposes of this project, the certificates will be self-signed.

[3] https://github.com/redhat-qe-security/SC-tests/blob/b29878ada72afd4da297c48419bdd62c9bc6d300/Local-user/test_local_user_login.py

2.6 Windowing systems in Linux

This section covers the most important components of windowing systems in Linux. The main focus will be display servers and their protocols. These will be important for the project, since the GUI testing frameworks can be dependent on a specific protocol.

Today, there are two prevalent display protocols, namely X [47] and Wayland [14]. X protocol was developed by Project Athena at MIT and it was first released in 1984. As the time passed, new versions of X protocol were released to cover the changing requirements. However, some new requirements were very problematic or impossible to implement, which raised the need for a new display protocol.

In 2008, Wayland protocol was introduced with the aim to be easier to develop, extend and maintain. The adoption had been relatively slow, because the protocol is incompatible with X. As of 2023, Wayland is used as the default in most Linux distributions. Notably, RHEL uses Wayland as default since version 8, released in 2019.

2.6.1 Display server

Display server [14] (also known as window server) is a program that coordinates the input and output between its clients and the rest of the operating system. It is a crucial part of any windowing system.

In context of display server, a client is any program that wants to draw on the screen. The client communicates with the display server via display server protocol (like X11 or Wayland). The protocol provides a standardized way for the programs to show graphical output. The client also receives events from the input devices like mouse or keyboard.

On behalf of the clients, display server interacts with the rest of the operating system. For example, this can be Kernel Mode Setting (KMS) for setting up the screen and *evdev* subsystem for reading the input devices. The purpose of the display server is to abstract these specifics of the operating system. Some of these resources can also be exclusive, meaning that they can be controlled by only one process. For example, the screen's framebuffer is an exclusive resource. Display server enables multiple applications to share the screen and the input devices.

2.6.2 X11 protocol

The X11 protocol [29] was designed with extensibility in mind, which means that new features can be added without breaking the existing clients. Support for an extension must be added to both server and the client before it can be utilized. The core protocol also includes functionality that enables clients to inquire about the extensions that the server supports, enabling them to determine what functionality they can or cannot use.

X11 was designed to be network transparent, which means that the server and the client cannot be assumed to run on the same machine. All communications between the client an the server must take place over the network. This design decision was made, because running X clients on a network connected machine used to be common in the past. Nowadays, this feature is rarely used. Modern desktop environments also often rely on other inter-process communication mechanisms such as DBus. Thus, some features will not work properly when running over network. The network transparency also adds large amount of overhead. This resulted in X11 extensions that allow communication over a UNIX socket, if the client and the server run on the same machine. When the X protocol

was first designed, the X server was responsible for many tasks that are now handled by kernel or libraries. Some of these tasks include:

- Device drivers: In the early days of X, device drivers for graphics cards and input devices (such as mice and keyboards) were implemented as part of the X server. Today, device drivers are typically implemented in the kernel.
- Drawing: In the past X server was used to draw graphical primitives in the framebuffer. The original intention was that the drivers could have accelerated code paths for drawing. In modern systems though, the rendering is done by widget toolkits such as Qt or GTK+. The rendered framebuffer is then passed to the X server.
- Window management: X used to be responsible for window management, including moving, resizing, and minimizing / maximizing windows. This responsibility has largely been taken over by compositors. Compositor takes all the windows and creates a final image that is displayed on the screen. It can also add its own graphical effects in the process. The compositor communicates with the X server via X composite extension protocol (see figure 2.3). The problem is that compositor and X server duplicate a lot of the same functionality. For example, both compositor and X server work with the order of the windows. Compositor needs to determine which window is active and which windows are obscured to render them in the correct order. X server needs to know the windows order, so it can direct the input events to the corresponding client. Therefore compositor has to be constantly synchronised with the X server.

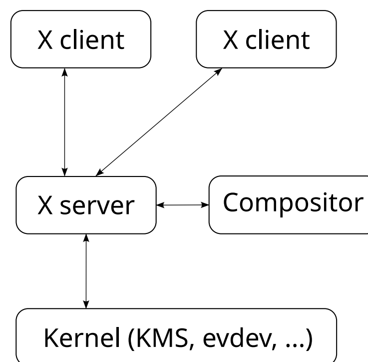


Figure 2.3: X window system architecture

Except for the basic functionality like displaying windows and reading input device events, X11 provides more features and extensions. For example clipboard or drag-and-drop are implemented by X11 extensions. Another X11 feature that is relevant to this work is screen capture. X11 enables any client to read the framebuffer of any other client. Although this could be considered a security vulnerability, it makes screen recording simple.

X.Org is the reference implementation of X11 server. Nowadays, X.Org is the only implementation that is widely used in Linux systems and is still actively maintained. Over

the time, X11 has accumulated a number of legacy features, which are rarely used in modern applications and make the maintenance difficult. This has raised the need for a new windowing system.

2.6.3 Wayland protocol

Wayland [14] is the new display protocol, which was designed with modern hardware and software in mind. Moving from X11, Wayland has removed many legacy features. By this virtue, both clients and servers are easier to implement and maintain.

The architecture of Wayland windowing system is also simpler and more streamlined than X11. Specifically, Wayland eliminates the need for separate display server and compositor (see figure 2.4). This design decision not only reduces complexity, but it also improves performance. In X11 architecture, the compositor needs to constantly synchronize with the X server when compositing the framebuffer. This introduces a considerable amount of context switching and thus the performance is reduced. In Wayland, compositor also fulfills the role of display server, which solves the issue of synchronization. There are numerous implementations of Wayland compositors. Weston serves as the reference implementation, and most desktop environments have implementations of their own. For example, GNOME desktop environment has a compositor called Mutter and KDE has KWin.

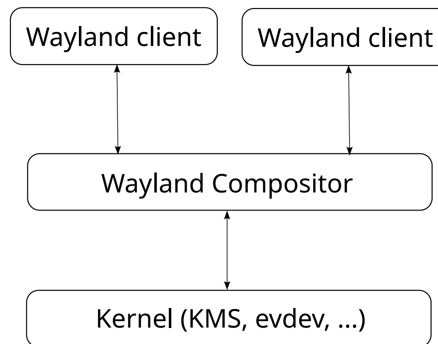


Figure 2.4: Wayland architecture [14]

Unlike X11, Wayland does not provide any drawing API. The compositor expects the clients to use any means to render into a shareable buffer. After the client has finished rendering, it notifies the Wayland server of the updated contents. In a typical app, the rendering is done by a GUI toolkit like GTK+ and Qt. The GUI toolkit may also use graphical acceleration to make the rendering more efficient.

Contrasted to X11, Wayland does not offer any kind of network transparency. Rather, it uses UNIX domain sockets and shared memory. UNIX domain sockets allow processes on the same machine to communicate with each other. This form of inter-process communication has less overhead than TCP/IP sockets. Framebuffers are allocated as shared memory.

One major difference in Wayland is its security model. Compared to X server, Wayland compositor runs as an unprivileged user. This reduces the potential impact, if the display

server was compromised by any of the clients. Each client has access only to its own window and its own set of inputs. This means that one client cannot access the framebuffer and inputs of another client, preventing malicious programs from spying on other clients. On the other hand, these security features make screen capture more difficult. More on screen capture in Wayland can be read in section 4.2.1.

Chapter 3

Existing solutions

Before developing the new GUI testing tool, existing solutions had been examined. Using an existing tool would avoid the significant time investment required to build a new tool from scratch. Various GUI testing tools are available and have established communities of users, making them a convenient option. However, commonly used GUI testing tools are not designed to test the display manager. The vast majority of the applications that are commonly tested run in a session. Therefore, the testing frameworks usually rely on the session. This becomes an issue when testing the display manager. Although we have not found any tools that are suitable for this project, the research still provides valuable insight into GUI testing tools.

3.1 Python with Accessibility API

Red Hat tests every component of RHEL, including some graphical programs. An interview was conducted with the Desktop quality engineering team to discover their approach. The approach uses accessibility API to expose the elements of the GUI. Additionally, the API provides a way of interaction with the elements. This approach has proven to be nonviable for testing GDM, but it has provided an insight into testing of GUI programs.

Before the test itself runs, graphical session on the system under test must be started. This is done using a script, which logs in as the target user and starts up the GNOME desktop environment. Then the testing script in Python is run. To read the state of GUI and control the system under test, only one library is needed. This library is called *Dogtail* [4] and it uses accessibility API [6] to interact with the system under test. All GUI elements are arranged in a tree structure. Using the library, the test script can find the elements in the tree by their name or other properties.

The API used for accessibility is named *AT-SPI2* [8] and it works over D-Bus. This protocol is implemented on the level of widget toolkit. Currently, both GTK and Qt support accessibility. Therefore, most of the graphical applications on Linux can be automated using *Dogtail* library. The problem with using accessibility API lies in testing the display manager. When the GDM is used, the session is not yet running. By logging into GDM (GNOME Display Manager), the session begins and the GNOME Desktop environment is started. Only after that, D-Bus can be used. For this reason, the approach is not suitable for this project.

3.2 openQA

The *openQA* [24] framework is capable of testing an operating system as a whole. It is a visual testing tool, meaning that it does not depend on any specific widget toolkit. *openQA* is most commonly used to verify the installation process of an operating system. It ensures that every new build of the distribution can be successfully installed. In order to run the tests, *openQA* creates virtual machines. The VMs are controlled with virtual keyboard and mouse to simulate the actions of a real user. To check the output of the system under test, *openQA* uses both the serial console and the virtual screen.

The testing tool *openQA* is free software and it is available under GPLv2 license. The project was started by the openSUSE for testing the nightly builds and it has been adopted by the Fedora community for the same purpose. Those distributions have official packages with *openQA*. In order to run *openQA* on other Linux distribution, the system has to have a Perl stack, *OpenCV* [23] library and QEMU virtualization.

A simple proof-of-concept test was implemented using *openQA*. Although main use case of *openQA* is installing the operating system from an ISO file, an image of an existing virtual machine can be used. In this case, the virtual machine was set up using QEMU and the qcow2 image was imported into *openQA*. Then a simple test case was performed. This test case would insert the virtual smart card, which was already installed on the machine. Then it would try to log in using GDM. Lastly, the screen of the system under test is compared to the image of unlocked desktop. This step determines whether the login was successful. The proof-of-concept was successful; however *openQA* was not chosen to implement other tests. This is because *openQA* is a complete solution for continuous integration. Using *openQA* in conjunction with the current Red Hat infrastructure would be difficult.

3.3 Visual GUI testing tool with VNC

Although there are many visual GUI testing tools available, we have not found any that could test the display manager. The issue lies in capturing the screen when the session is not running. None of the tools that were tested were able to capture the framebuffer from the display manager.

Although no GUI testing tool is able to capture the framebuffer, the issue can be circumvented by using a VNC server. The VNC server would share the framebuffer of the tested display manager and the tool would be connected over a network. The testing tool would run on the same machine as the VNC for the sake of simpler setup. There are several tools available that support testing over the VNC protocol. The VNC connection over network can also make the testing more transparent, which can simplify creating tests and debugging.

3.3.1 Available tools

SikuliX is a framework that allows automating visual workflows. The main use case of *SikuliX* is creation of automation scripts for graphical programs, but it is often used for automated testing. *SikuliX* is an example of visual GUI testing tool. It is able to visually detect the contents of the screen, move the mouse pointer and type in the text fields. *SikuliX* relies exclusively on image recognition; it does not have any insight into the internal structure of GUI. By this virtue, it can automate any testing of any program, without being dependent on a specific GUI framework.

Normally, *SikuliX* uses the real screen of the computer to run the tested program and emulates the mouse and keyboard to interact with it. However, the screen and input devices can be substituted with remote ones that are accessed via VNC protocol. Over VNC, *SikuliX* is able to connect to any computer to perform the tests The *SikuliX* scripts can be created and replayed with *SikuliX* IDE. However, the scripts are able to run independently in Python, Ruby or JavaScript.

3.3.2 VNC servers

There are many different ways in which VNC server can work. VNC itself is a network protocol that allows viewing remote framebuffer and sending mouse and keyboard inputs. The framebuffer can be virtual or it can correspond to a real screen. For the purposes of this project, both virtual and physical display are usable.

X11vnc

x11vnc [46] is a VNC server that allows sharing a real X display. A real X display corresponds to a physical monitor, mouse and keyboard. *x11vnc* communicates with X server via X11 protocol to get the framebuffer. This means that it cannot work with Wayland compositors, which was also confirmed by testing. This solution was tested using the command below.

```
sudo -u gdm x11vnc -auth /run/user/42/gdm/Xauthority -display :0
```

The framebuffer can be accessed, but this approach also has some drawbacks. Although GDM can be accessed and controlled, the screen sharing stops working after the user signs in. A probable reason for that is the `Xauthority` file, which has to be different in the session. This makes the solution quite inconvenient, since the command for sharing is specific to every user.

According to documentation, *x11vnc* can be used in a more low-level way that does not involve communication with X server. Sharing a raw framebuffer can be accomplished using commands below.

```
x11vnc -rawfb console
x11vnc -rawfb map:/dev/fb0
```

When tested, the displayed screen was black, but the input devices were working. The contents of the screen could not be accessed, since display servers in modern systems do not use the framebuffer device. Framebuffer device is now only used for virtual terminals, which were visible in the shared framebuffer. Nowadays, all graphical interfaces are graphically accelerated, which means they have to draw using different interfaces.

3.3.3 TigerVNC with XDMCP

TigerVNC [45] is another open-source implementation of VNC protocol. TigerVNC provides both client and server application, and some other command line utilities. The server for UNIX-like systems is called *Xvnc*. This program contains both VNC server and an

X server. The X server is completely independent from the system's display server. *Xvnc* also creates virtual screen and input devices, which are completely separate from the real ones.

Logging into the system via GDM over VNC is officially supported use case for this software. This is done using XDMCP, which stands for X Display Manager Control Protocol. XDMCP works over a network, it uses UDP port 177. Using XDMCP, X server can request the display manager to start a session. In this case *Xvnc* requests GDM to start a session. In order for this to work, XDMCP needs to be enabled in GDM configuration file (`/etc/gdm/custom.conf`). For one session, *Xvnc* can be run using the command below. *Xvnc* can be also set up to run on demand for every connection [2]. This can be achieved using systemd socket activation.

```
Xvnc -inetd -query localhost -geometry 1920x1080 -once \
    -SecurityTypes=None
```

This approach was was tested successfully. *Xvnc* allows user to log in using GDM over VNC protocol. When the user logs in, GNOME session is started and it can be controlled over the same VNC connection. However, it was discovered that GDM behaves differently when using XDMCP. Smart card authentication did not work, when GDM was accessed over VNC. This makes the approach unsuitable for the project.

Chapter 4

Design of a new solution

Since none of the existing solutions yielded satisfactory results, a new one had to be created. First, the requirements were specified.

In section 3.1, it was already established that accessibility API cannot be used to test the display manager. Therefore, the new solution has to perform visual testing. To create a visual GUI testing framework, there are three main problems to solve. First, the framework has to capture the screen (i. e. access the framebuffer). After the screenshot is captured, the framework has to recognize GUI elements in the image. The last problem is to control the system under test. This means injecting mouse and keyboard input into the system.

4.1 Requirements

After examining the abilities of existing GUI testing tools, the requirements for the new solution have been established. The requirements are as follows:

- The new tool must be able to test GDM, GNOME lock screen, and lock on removal feature in GNOME desktop environment. Automated tests are required to test all the components that have been tested manually.
- The solution cannot create virtual machines. This requirement is based on the infrastructure used in Red Hat. Before the test is run, the system under test is automatically provisioned as outlined in section 2.5.2. The testing script is then run within the provisioned machine. The infrastructure is set up in a way that the provisioned machine should be the system under test. It is running the version of RHEL that needs to be tested. It also has pre-installed packages according to the test's metadata. Creating a virtual machine from the script would be possible, but setting it up would be very difficult.
- The tool has to be independent of X11 protocol, since RHEL 8 and 9 use Wayland by default. X11 is still supported in RHEL 9, but it is deprecated [34], meaning that it should no longer be used in new projects. Considering that the testing framework should be supported long-term, it cannot depend on X11.
- The solution has to be implemented in Python. Python was chosen for its versatility and its vast collection of libraries. Using the libraries from PyPI (Python Package Index), implementation can be simplified substantially. *SCAutolib*, which is used for

automated smart card testing, is already written in Python. Writing the GUI testing solution in Python makes it easy to integrate into *SCAutolib*.

4.2 Accessing the framebuffer

The framework has to read the screen in order to detect the graphical elements. In section 3.3.2, it was already established that no VNC server is viable for this usage. The capturing of the screen can be done in multiple ways. The screen can be captured using the Wayland compositor or directly via KMS interface [43].

4.2.1 Using Wayland compositor

Most applications use functionality of the X server or the Wayland compositor to capture the screen. With X server, capturing the screen is quite simple, since sharing the framebuffer is part of the X11 protocol. However, the Wayland protocol does not provide an API for taking screenshots nor screen sharing [7]. For this reason, screen sharing in Wayland is quite fragmented.

Some Wayland compositors, like Weston or Mutter, have implemented their own API for screen sharing. The API specific to Mutter is named org.gnome.Shell.Screenshot and it works over D-Bus. Considering the requirements, using API specific to Mutter would be acceptable.

There is also *xdg-desktop-portal* API, which supports taking screenshots and screen sharing. This API was developed by Freedesktop.org and is supported by most of the Wayland compositors including Mutter. Xdg-desktop-portal API works over D-Bus.

There also exists Wayland extension for taking screenshots named *wlr-screencopy*. This extension is implemented in Wayland compositors based on *wlroots* library. However, the extension is not supported in Mutter, so it cannot be used for this project.

From the existing APIs, Mutter supports its own *org.gnome.Shell.Screenshot* API and *xdg-desktop-portal*. Nonetheless, both APIs work over D-Bus, which cannot be used before the session has started. For this reason none of the APIs can be used to capture the screen, when using GDM. No API was found that satisfies the requirements.

4.2.2 Using Kernel Mode Setting (KMS)

Besides using Wayland compositor, there is another way to capture the screenshots. The framebuffer can be accessed directly using Kernel Mode Setting (KMS) [43]. This method is very low-level; it does not depend on X server nor the Wayland compositor. When using KMS, kernel uses the GPU driver to download the framebuffer from GPU hardware.

A tool has been discovered that has the capability to capture a screen over KMS interface. Its name is *ffmpeg* [5] and it is commonly used for recording, converting and streaming audio and video. This utility comes from FFmpeg suite, which consists of multiple libraries and utilities for handling multimedia. The *ffmpeg* tool has an input device named kmsgrab, which is able to read framebuffers via KMS interface. This solution has proved to work both on bare metal and virtual machines. Additionally, it meets the all the requirements for the solution, so it will be used for the implementation.

4.3 Methods of screen contents recognition

Every visual GUI testing tool needs to implement image processing to recognize the graphical elements. The matching can be done in multiple ways, this work explores template matching [22] and OCR [38]. Both of the methods are suitable for the project, either one or both of them could be used. In the end, OCR was selected for reasons listed in sections below.

4.3.1 Template matching

Template matching works by comparing the tested image to a reference. This principle is utilized in other VGT tools like *openQA*. Python can do template matching using various libraries, for example using *OpenCV*. Template matching method can locate a template image in a larger image. In VGT tools, the template is usually image of the required graphical element and it is located in the screenshot.

By its principle, template matching does respond to any changes in graphical design. When the font or the background color changes just slightly, the template images have to be updated. This can result in tests that require more maintenance than tests using OCR. Template matching is also dependent on resolution of the screen. If the resolution changes, some graphical elements will be scaled and will not be detected. On the other hand template matching is better for locating icons, which contain no text. These cannot be found using OCR. Template matching is typically lighter on the hardware resources than OCR.

4.3.2 Optical character recognition

Optical character recognition (OCR) works by recognizing characters and words from the image. OCR can be done in Python using Tesseract, which is a popular open-source OCR engine.

Compared to template matching, OCR has the advantage of being independent of the graphical design. If the font or color changes, OCR will still work which is not the case of template matching. OCR is also independent of the screen resolution. This removes the need to the same resolution on every system under test. OCR also has some disadvantages compared to template matching, but most of them can be evaded.

As an example, OCR cannot be used for locating icons. However, most of the programs can be controlled just using buttons with text. If there are no such buttons available, keyboard shortcuts can be used. Another potential disadvantage of OCR can be the accuracy. When using OCR to digitize scanned documents, some letters may be detected incorrectly. Nonetheless, this poses no problem when scanning screenshots. Compared to scanned documents, screenshots have no artifacts like image grain. The text in screenshots is is also perfectly horizontal. These properties make the screenshots easily readable for OCR engine. This has been confirmed through manual testing by the author.

All things considered, OCR seems to be the better option for the project. The main deciding factor was the amount of maintenance needed. In this area, OCR has a clear advantage.

4.4 Emulation of user input

To control the system under test, a mouse and a keyboard is needed. This section is about finding Python libraries that would emulate these input devices. Although there are many Python libraries to control mouse and keyboard, some of them depend on X protocol.

4.4.1 Mouse

The first library that was tested was mouse [20] from PyPI (Python Package Index). This library can both read mouse events and send mouse events. According to the documentation, it works on both Linux and Windows. The library is written in pure Python, so it does not require compilation on the target system. The API is simple and high-level; it supports moving the mouse in absolute coordinate system. However, library's documentation does not mention Wayland support. Based on our testing, this library does not work under Wayland, so it cannot be used.

Another library that was examined was *python-uinput* [31] from PyPI. This library is very low-level compared to mouse. It is written partly in C and thus needs to be compiled during the installation. *python-uinput* interfaces directly with the *uinput* kernel module [44]. This is done by writing into `/dev/uinput`, which is then read by the kernel. *python-uinput* is only a simple binding for *uinput* API. *uinput* kernel module can create not only virtual mouse, but also any other event device. This can be a keyboard, gamepad or a graphical tablet. According to our tests, the library functioned correctly under both Wayland and X.

4.4.2 Keyboard

For controlling the keyboard, keyboard [17] library from PyPI was chosen. The library is written purely in Python and it allows both listening to and sending keyboard events. The API is high-level, which makes the implementation straightforward. For example, the library has function `keyboard.write(str)`, which types the string passed as an argument. Another example is `keyboard.send(key)`, which can press keys like enter or escape. The library was tested successfully.

4.5 Integration into SCAutolib

This section explains the reasons why the new GUI testing framework is implemented as a module in *SCAutolib*. With the tests being written in Python as separate files, it makes sense to have one module for graphical testing. This module would be imported in every test that needs to interact with the system's GUI.

The module for graphical testing combines the functionality of capturing screen, doing the image processing and controlling the system under test. It offers a high-level API, so that the graphical tests can be as simple as possible. An example of a function in this API is `click_on(text)`. This function takes a screenshot, runs OCR algorithms, finds the given text in the screenshot and then controls the mouse to click on the correct coordinates. Everything will happen internally in the function.

The Graphical User Interface of the system is a stateful system. The tests always have to start in a pre-defined state of the GUI. Python language offers context managers which are well suited for this problem. Specifically, context managed class can keep the internal state and have methods to interact with the system. Then, the testing script itself has to import only one class.

The new module for graphical testing could be integrated into *SCAutolib* or it could be made an independent library. A decision was made to make the module a part of *SCAutolib* for the following reasons. Firstly, the module will always be used in conjunction with other parts of the library, so it makes sense to have it integrated. Secondly, *SCAutolib* already has scripts for installing RPM packages during the setup phase, which will be useful for installing dependencies such as *tesseract* and *ffmpeg*.

Chapter 5

Implementation

This chapter explains the implementation details of the project. The whole project is implemented in Python. The only exception are the shell scripts to run the tests in Red Hat infrastructure. First, a proof-of-concept was implemented to demonstrate the feasibility of the solution. Then, GUI related functionality was implemented in `gui.py` and checking logs in `log.py`. Both of these files were added to *SCAutolib*.

After finishing the API of new modules, GUI tests have been written. Tests are split into two files; one contains tests for GDM, while the other has tests for GNOME lock-screen and lock on removal feature. The whole implementation (new modules in *SCAutolib* and new tests) counts 797 lines (348 lines of code, 304 lines of comments). The lines were calculated using *cloc* tool. The code can be found in SCAutolib repository[1] and SC-tests repository[2] on GitHub.

5.1 Environment used for development

For the development of this project, a virtual machine (VM) running RHEL 9 was set up. This version was chosen, since it will be target system for all the tests. The VM was created with *virt-manager* software using QEMU backend. *virt-viewer* was used to interact with to interact with the machine's GUI. A shared folder was set up using NFS (Network File System), so that the VM has access to python scripts developed on host. To control the virtual machine, SSH connection as the root user[3] was used. Each library used in implementation was tested interactively from IPython shell.

5.2 Screen capture using ffmpeg

The screen capture is implemented using *ffmpeg* with the kmsgrab [5] input device. According to *ffmpeg* documentation, *ffmpeg* can be run as in example below to capture and save video from kmsgrab.

```
ffmpeg -f kmsgrab -i - -vf 'hwdownload,format=bgr0' output.mp4
```

[1] https://github.com/redhat-qe-security/SCAutolib/

[2] https://github.com/redhat-qe-security/SC-tests/

[3] Generally, SSH connection as the root user is considered a bad security practice. In this case though, the virtual machine is connected to a virtual network that can be accessed only by the host.

In order for this example to work, the process has to run as DRM master or with `CAP_SYS_ADMIN` capability. In practice, the command will run as root which bypasses all the checks for capabilities. Kmsgrab input has some options to specify the captured framebuffer and its format. For example, if the system has more than one GPU, the required GPU can be set using `device` option. By default, the device is set to `/dev/dri/card0`. The framebuffer can also contain pixels in different formats. Kmsgrab can detect the format automatically in Linux 5.7 or later, but it needs to be specified for older versions. With this example working, the next step was capturing only one image. This can be done using `-frames` option. Additionally, some options were added to suppress irrelevant logs.

```
ffmpeg -hide_banner -y -f kmsgrab -i - -vf 'hwdownload,format=bgr0' \
    -frames 1 -update 1 image.png
```

Capturing the screenshots is implemented in `Screen` class. Method `screenshot` captures the screenshot and returns its filename. An instance `Screen` also contains an attribute with screenshot number, which is incremented with every screenshot. The init function of the class takes a parameter to specify the folder, in which the screenshots will be saved.

5.3 Screen contents recognition

After the screenshot is captured, the GUI elements need to be recognized using OCR. This process is quite complex and is divided into multiple stages. First, the image is read and preprocessed using *OpenCV* library [23]. Then it is converted into text data using *Tesseract*. Finally, the data from *Tesseract* is manipulated by *pandas*, and the coordinates of elements are calculated.

5.3.1 Preprocessing

For the best results of optical character recognition, the images need to be processed beforehand. *OpenCV* (Open Source Computer Vision Library) provides many algorithms and is widely used for image processing. The algorithms are implemented in C++ and they are highly optimized. *OpenCV* also provides bindings for Python for easier implementation.

The recommended input format for *Tesseract* OCR is binary image [40]. It means that the image only uses two colors, usually represented as black and white. After the screenshot is read from file, it is converted into grayscale. Next the image is the upscaled and the thresholding function is applied. After these preprocessing steps, the image is ready for optical character recognition.

Image is converted to grayscale using `cv2.cvtColor` function, which can be used to convert image into any color space. This conversion removes information from the image, but for purposes of this project, reading colorful text is not important. If the GUI is well-designed, the text should be readable in grayscale.

Next, the image is upscaled using `cv2.resize` function. Upscaling is not strictly necessary, but it significantly improves the results for small fonts. This is demonstrated in figure 5.1. The uppermost image shows a detail of a screenshot taken in RHEL 9. The middle image shows how the processed image looks without upscaling. The lowermost image shows the processed image which was upscaled by a factor of two. It can be observed that

the image that was upscaled before thresholding is more detailed. The edges of the fonts are also less pixelated. This leads to better results of OCR.

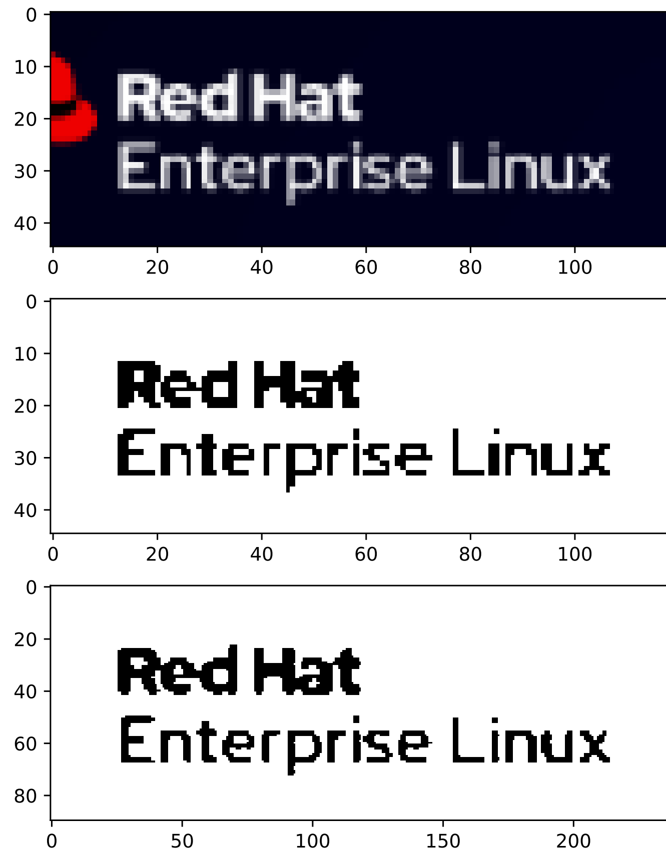


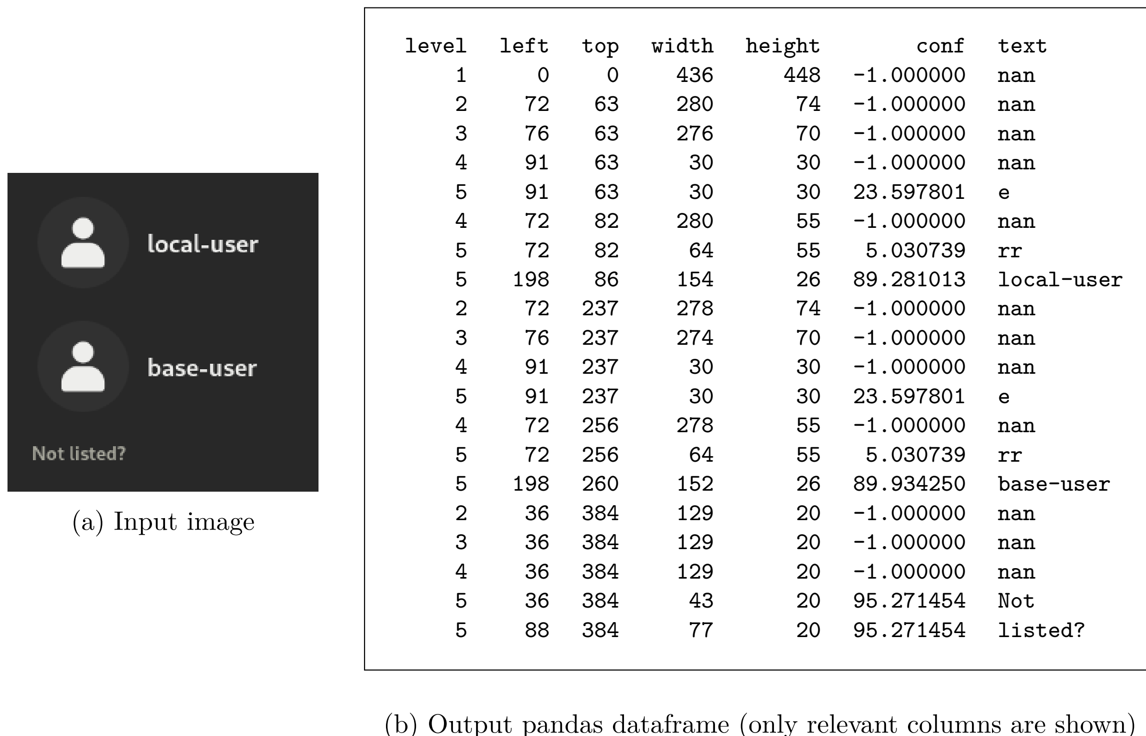
Figure 5.1: Comparison of original screenshot (top) to a preprocessed image in original resolution (middle) and to an upscaled preprocessed image (lowermost). Units of both axis are pixels.

The last step of preprocessing is the thresholding implemented by `cv2.threshold` function. For purposes of this project, thresholding with constant threshold value is sufficient. If text on a gradient background had to be detected, adaptive thresholding would be a better option. The code used for preprocessing the screenshot is shown in the example bellow.

```
grayscale = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
upscaled = cv2.resize(grayscale,
                      dsize=None,
                      fx=UPSCALING_FACTOR,
                      fy=UPSCALING_FACTOR,
                      interpolation=cv2.INTER_LANCZOS4)
_, binary = cv2.threshold(upscaled, 120, 255, cv2.THRESH_BINARY_INV)
```

5.3.2 Optical character recognition

Pytesseract binding offers functions to convert image into multiple different formats. The most suitable function for this project is `image_to_data`. This function takes the image and converts it directly to *pandas* [28] dataframe as shown in figure 5.2.



(a) Input image

level	left	top	width	height	conf	text
1	0	0	436	448	-1.000000	nan
2	72	63	280	74	-1.000000	nan
3	76	63	276	70	-1.000000	nan
4	91	63	30	30	-1.000000	nan
5	91	63	30	30	23.597801	e
4	72	82	280	55	-1.000000	nan
5	72	82	64	55	5.030739	rr
5	198	86	154	26	89.281013	local-user
2	72	237	278	74	-1.000000	nan
3	76	237	274	70	-1.000000	nan
4	91	237	30	30	-1.000000	nan
5	91	237	30	30	23.597801	e
4	72	256	278	55	-1.000000	nan
5	72	256	64	55	5.030739	rr
5	198	260	152	26	89.934250	base-user
2	36	384	129	20	-1.000000	nan
3	36	384	129	20	-1.000000	nan
4	36	384	129	20	-1.000000	nan
5	36	384	43	20	95.271454	Not
5	88	384	77	20	95.271454	listed?

(b) Output pandas dataframe (only relevant columns are shown)

Figure 5.2: Conversion of a screenshot to a dataframe using Tesseract

The dataframe is in table format [36], where each recognized structure is represented with one line. The structure can be a page, block, paragraph, line or a word. Each structure in dataframe has the following properties:

- `level`: Indicates, whether this structure is a word, line, paragraph, etc.
- `page_num`: The number of the page in document.
- `block_num`: The number of the block within page.
- `par_num`: The number of the paragraph within block.
- `line_num`: The number of the line within paragraph.
- `word_num`: The number of the word within line.
- `left`: The x coordinate measured in pixels from the top of the image.
- `top`: The y coordinate.
- `width`: Width of the structure in pixels.

- `height`: Height of the structure in pixels.
- `conf`: Confidence of the OCR engine as percentage.
- `text`: The detected text.

Only word structures (level = 5) have `text` property, since content of the higher level structures can be obtained by concatenating words. Confidence values are also exclusive to words, other structures have confidence value of -1 as shown in subfigure 5.2b. In the implementation, `text` attribute is used to find the element. Then the properties `left`, `top`, `width` and `height` are used to calculate item's coordinates.

5.3.3 Searching in dataframe

After the screenshot is read using OCR, the output is stored as a *pandas* [28] dataframe. *pandas* library is designed to manipulate datasets. Searching in *pandas* dataframe is much simpler than in Python built-in data structures. In this project, the most common usage of *pandas* is finding the entry with a specific text. This can be done using fancy indexing in *pandas* as shown in the example below. The example shows selection of rows in dataframe, where `text` attribute is equal to `key` variable.

```
selection = df[df['text'] == key]
```

5.4 Keyboard

Keyboard is implemented using the keyboard library from Python Package Index. The GUI class from the new module has methods `kb_write` and `kb_send`, both of which are wrappers for methods from keyboard library. The reason for integration of the keyboard library is that it has some bugs that need to be worked around. Workarounds are implemented in GUI class's init method and in the wrappers. For example, `keyboard.write` function does not work, unless `delay` parameter is set to value larger than 0. `kb_write` wrapper gives it default value of 0.1 to work around this problem.

This makes implementation of the tests much simpler, since the programmer does not have to keep in mind the specifics of all the libraries. The test scripts can also be more concise and readable, which is the main goal of *SCAutolib*.

5.5 Mouse

The mouse input was implemented using uinput library, since no suitable high-level libraries were found. Hence, the high-level API was implemented as a part of the project. The implementation is contained in the `Mouse` class.

5.5.1 Uinput library

In order to emulate input devices, uinput kernel module must be loaded beforehand. This can be done with the `modprobe` command. The command is run in the mouse's init method before the uinput library is used.

Since uinput can create more devices at once, each one has to be created with constructor `uinput.Device`. This library can emulate any device including mouse, keyboard, gamepad or a graphical tablet. This is why the constructor takes a list of all the device's buttons and axis. A constructor for a basic mouse is shown as the example below. Note that some mice may have extra buttons, but those are not needed for purposes of this project.

```
self.device = uinput.Device([
    uinput.REL_X,
    uinput.REL_Y,
    uinput.BTN_LEFT,
    uinput.BTN_MIDDLE,
    uinput.BTN_RIGHT,
    uinput.REL_WHEEL,
])
```

When a graphical element is detected on the screen, its coordinates are absolute. High-level mouse libraries can communicate with the X server or Wayland compositor to move the cursor to exact coordinates. Uinput can move the cursor in both relative and absolute coordinates. These options are discussed in the following sections.

5.5.2 Relative coordinates

Moving the mouse cursor in relative coordinates most closely represents a real user interacting with the computer. In libinput, the emulated mouse behaves exactly the same way as a real one. However for this project, the mouse cursor has to be moved to absolute coordinates.

The simplest solution is to move the cursor into one corner of the screen and then move it to the position. The most sensible corner to move to is the top left corner, since its coordinates are (0, 0). Then the mouse is moved the desired amount in both axis. The downside of this approach is that it depends on a specific mouse sensitivity. Mouse acceleration also has to be turned off. This solution does work in RHEL 9 and it was successfully used in the proof of concept.

A better solution could use template matching to find the cursor in the image. Then a feedback loop can be implemented to move the cursor into desired position. This would work in the in same way that a human controls the mouse cursor. Major downside of this solution is the speed and complexity. A screenshot has to be taken after every movement of the cursor. Then the template matching is performed and a difference in position is calculated. This process iterates, until the cursor is in the desired position.

5.5.3 Absolute coordinates

Using absolute coordinates solves most of the problems of moving in relative coordinates. This method of input is used for example with touchscreens, since their output is in absolute coordinates.

When the uinput device is initialized, minimum and maximum coordinates of each axis are given. This effectively defines the resolution of the input device. The resolution is not related to the screen resolution, libinput normalizes the coordinates and maps them to the

screen. This solution is used in the project, since it's simpler and more reliable that using relative coordinates.

5.6 Test setup

The tests for GUI were developed on a virtual machine, which had been set up manually. The setup involves installing RPM dependencies, pip dependencies and configuring the GNOME shell. To run the tests automatically on Red Hat infrastructure, the setup needs to be automated as well. The general setup of system under test is already implemented in *SCAutolib*, so our work comes down to extending the existing scripts.

The setup for system under test is done with `scauto` command. An option `--graphical` was added, because some tests do not require the dependencies for GUI testing. The dependencies for graphical testing are also quite large in size, so they should be installed only if necessary.

5.6.1 RPM dependencies

The new GUI testing module depends on two additional RPM packages, namely *tesseract* and *ffmpeg*. *SCAutolib* already implements checking and installing RPM packages, so implementation comes down to adding the packages. The packages are checked and installed in module named `controller.py`.

Tesseract package is available in RHEL repositories making the installation simple. Although *ffmpeg* is free software distributed under LGPL licence, it is not present in RHEL or Fedora repositories. The reason is that *ffmpeg* contains some codecs for patented media formats [35].

ffmpeg can be installed from RPM Fusion, which is a community repository. It provides the packages that are not distributed by Red Hat for various reasons. This solution is not ideal though, since packages from RPM Fusion cannot be installed while provisioning the machine.

Alternatively *ffmpeg-free* can be used. This package is located in EPEL (Extra Packages for Enterprise Linux). *ffmpeg-free* package provides the same *ffmpeg* utility. The only difference is that *ffmpeg-free* is compiled without the patented codecs. For capturing the screen, *ffmpeg-free* works in the exact same way, since none of the patented codecs are used. This solution is preferable to RPM Fusion, since EPEL package can be installed while provisioning the system under test.

5.6.2 PyPI dependencies

In *SCAutolib*, PyPI dependencies are stored in `requirements.txt`. Although dependencies for GUI testing could be added here, they would be installed even if not needed. For this reason, the packages were added as optional dependencies directly into `setup.py`. *SCAutolib* with dependencies for graphical testing can be installed using command below.

```
pip install SCAutolib[graphical]
```

5.6.3 GNOME desktop setup

When the tests were run on a machine in Red Hat infrastructure, the first test always failed. After closer inspection, the screenshots showed „Welcome to RHEL“ dialogue window. This was not a problem on local virtual machine, since the window appeared only once and every subsequent run of tests succeeded. To fully automate the testing, the GNOME desktop environment needs to be set up first.

On local virtual machine, it was found that the welcome dialog appears only when a new user is created. Deleting user's `~/.config` folder also causes welcome dialog to appear. The parameter responsible for showing the dialog was determined to be saved in dconf database. Dconf [41] is a simple key-based configuration system. It is mainly used by GNOME apps for storing their settings for each user. The dconf database was dumped by running `dconf dump /`.

```
[org/gnome/shell]
welcome-dialog-last-shown-version='40.10'
```

If the version in the entry above is higher or equal to the version of GNOME currently installed, the dialog is not shown. Setting this value to a large number ensures that the dialog will never be shown. Ideally, the setup should disable the dialog for all users. This can be done by using custom default values. Default values can be set by creating config files in `/etc/dconf/db/local.d/`. In the setup phase in *SCAutolib*, the file below is copied into the directory.

```
[org/gnome/shell]
welcome-dialog-last-shown-version='100.0'
```

Then, `dconf update` is run to update the database from the system files. After the user is created, the welcome dialog does not appear anymore.

Another problem to solve was a subscription notification. This notification appeared at the top of the screen after signing in. It did not cause any proof-of-concept tests to fail, but it could potentially affect other tests. The notification can be disabled by masking its user systemd unit. To deactivate the notification for all users, `systemctl` command can be run with `--global` option.

```
systemctl --global mask org.gnome.SettingsDaemon.Subscription.target
```

5.7 Reading system logs

Most of the graphical tests verify the state of the system by reading the contents of the screen. However, in case of some tests, the state of the system cannot be completely determined from the screen. An example of this test is an unsuccessful login into the system. In that case, the screen shows GDM password prompt with no clear indication, whether there was an attempt to log in. For this reason, a module for reading the logs was implemented.

The main idea behind reading the logs for testing is that some actions in GUI will be logged. The log module has to provide a way to assert that an action has caused a specific log to be written. This assertion has to be independent of previous logs in the file.

There are two basic ways of reading the logs. The first way is reading the log files, which are usually stored in `/var/log`. Each program has its own file, where its logs are stored. The log files should be present on any UNIX-like system. The other way of reading the logs is using systemd journal. Systemd journal is the modern approach to collecting logs. All logs are in the same format and they are stored centrally. To implement the log reading functionality, the first approach was chosen. Reading log files is trivial in Python and it does not require any new libraries.

```
with assert_log('/var/log/secure', r'.*authentication success.*'):
    gui.kb_send('enter', wait_time=10)
```

The log module provides just one context manager to verify that a log was added. The context manager has two parameters. They specify the log file and a regular expression to search for. When the context manager enters, the log file is opened and the file pointer is moved to the end of the file. This assures that no previous logs will be read. Then the actions within the context managers are run. In case of the example, a keyboard event is sent, which should cause an authentication in GDM. When the context manager exits, the newly added content of the log is read. If a line in the log matches the regular expression, the context manager exits successfully. Otherwise an exception is raised and the test fails. The log module is well suited for for both CLI and GUI tests. It does not add any extra dependencies to *SCAutolib*.

5.8 GUI module's API

The GUI module provides a high-level API, so that the tests can be as simple as possible. All methods, that are needed for the tests are implemented within the GUI class.

- `click_on(key, timeout)` finds and clicks the matching word on the screen. When this function is called, it takes a screenshot and converts it into *pandas* dataframe. Then, it tries to find the key in the dataframe. If the key is found, mouse is moved to the position of detected element and click event is sent. In case none of the recognized string match, the process is repeated until the time runs out. Then, an exception is raised, causing the test to fail. Timeout argument is supplied in seconds, default value is set to 30 seconds.

- `kb_write(*args, **kwargs)` types the string passed as argument on an emulated keyboard. It is a wrapper for `write()` method from keyboard library. `kb_write` also implements a workaround for a bug in the keyboard library.

- `kb_send(*args, **kwargs)` is used to press shortcuts on the keyboard. It is a wrapper for `send()` method from keyboard library.

- `assert_text(key, timeout)` works in the same way as `click_on`, except it does not do anything, after the key is found. If the key is not found within the specified timeout, an exception is raised.

- `assert_no_text(key, timeout)` asserts that the screenshot does not contain given key. This function raises exception, if the key is found in screenshots within the specified timeout. The default value of timeout is 0, which means that only one screenshot will be taken.

All GUI methods implement logging, so that the GUI test can be debugged from logs. The filename of every taken screenshot is also included in the log. In case something goes wrong during the test, the screenshots can be inspected and correlated with the actions.

The methods `click_on`, `kb_write` and `kb_send` automatically take screenshots before and after the action. If these screenshots are exactly the same, an exception is raised. The reason for this behavior is that a mouse click or a keyboard action should almost always cause a visual response. This functionality is implemented using a decorator in `action_decorator`.

Outside of GUI class, the module exposes `Mouse Screen` classes. These are used internally in GUI class; importing them in tests is not necessary. Functions `image_to_data` and `images_equal`, which convert image to dataframe and compare two images, can also be used from outside the module.

5.9 Test cases

After the library was finished, test cases were implemented. The existing test cases for CLI are using pytest framework. This allows using fixtures for the users or parametrization of the system configuration. The new test cases for GUI will also utilize pytest framework.

The test cases were designed to cover all the features that should be tested manually (described in section 2.5.1). The test cases are split into two python modules. The first one tests logging into the system using GDM. The other one contains test cases for the GNOME lock screen and lock on removal feature.

The tests for GDM cover logging in with both password and a smart card. For each method, there is a test case for successful and unsuccessful authentication. The authentication should be tested for different *SSSD* configurations. *SSSD* configuration specifies, whether the smart card is required or optional. If both configurations behave the same way, the test case can be parameterized using a decorator from pytest. This prevents unnecessary code duplication. The implemented test cases listed below. The test steps of each test case can be found in appendix A.

- `test_login_with_sc`: The user logs in using smart card with the correct PIN. The case is parametrized for the smart card being optional and required.
- `test_login_with_sc_wrong`: The user tries logging in using smart card with an incorrect PIN. The case is parametrized for the smart card being optional and required.
- `test_login_password`: The user logs in using his password. Smart card parameter is set to optional, since logging in with password with smart card required is not possible.
- `test_login_password_wrong`: The user tries logging in with an incorrect password. Smart card parameter is set to optional.
- `test_insert_card_prompt`: Smart card parameter is set to required. The user tries to log in with GDM before inserting card. GDM should prompt the user to insert card. After the card is inserted, user log with the correct PIN.

- `test_lock_on_removal`: The user removes the card while logged in to lock the screen. The screen should lock automatically. After waking up the screen and re-inserting the card, user can unlock the session with the card's PIN. Lock on removal parameter is set to enabled. The case is parametrized for the smart card being optional and required.

- `test_lock_on_removal_password`: The user inserts and removes the card while logged in with password. Nothing should happen, since the user has started the session with password authentication. Lock on removal parameter is set to enabled. Smart card parameter is set to optional, since the user is logged in with password.

- `test_lockscreen_password`: The user logs in with password, inserts the card, and locks the screen manually. The screen should be unlocked with password, even though smart card is inserted. Screen should be always unlocked with the same authentication method that was used to start the session. This case is parametrized for lock on removal parameter.

The code below is a running example, specifically the `test_login_with_sc` test case.

```
# pytest will automatically run this test case with required parameter set to
↪ True and False
@pytest.mark.parametrize("required", [(True), (False)])
def test_login_with_sc(local_user, required):
    # Prepare regular expression that should be found in /var/log/secure
    expected_log = (
        r'.* gdm-smartcard\]\[[0-9]+\]: '
        r'pam_sss\(gdm-smartcard:auth\): authentication success;'
        r'.*user=' + local_user.username + r'@shadowutils.*'
    )

    # Enter context manager for authselect, smart card and GUI
    with Authselect(required=required), local_user.card(insert=True), GUI() as
    ↪ gui:
        # After GDM is started and smart card is inserted, PIN prompt should
        ↪ appear automatically
        gui.assert_text('PIN')
        # Fill in the user's PIN
        gui.kb_write(local_user.pin)

        # Start reading /var/log/secure
        with assert_log('/var/log/secure', expected_log):
            # Press enter on emulated keyboard and wait for 20 seconds
            gui.kb_send('enter', wait_time=20)
            # Context manager exits, /var/log/secure is searched for the
            ↪ expected log. In case the log is not found, an exception is
            ↪ raised

        # When the user logs in successfully, GNOME shell will show
        ↪ 'Activities' in the top left corner of the screen
        gui.assert_text('Activities')
```

5.10 Running the tests in Red Hat infrastructure

To run the tests in Red Hat infrastructure, test metadata (in form of Makefile) and entry point shell script had to be created (see 2.5.2). *SCAutolib* also requires a json file to set up the authentication on the system. This file contains the users, their passwords and their smart cards (see 2.5.3). The files were copied from existing CLI tests. The contents of files are shown in appendix B.

In the `Makefile` RPM dependencies that are needed only for graphical tests were added. In the entry point script (`runtest.sh`) installation of ffmpeg-free was added. Packages from EPEL repository (Extra Packages for Enterprise Linux) cannot be installed while provisioning the machine, so they have to be installed from the script. The sections have been rewritten to run the new test cases. The tests were then run in Red Hat OpenStack using *1minutetip*, which is a tool internally used in Red Hat.

Chapter 6

Evaluation of implemented solution

Before the solution was deployed, various performance metrics have been measured. Specifically, the time for taking a screenshot and using OCR was measured. After the deployment, the time of running the whole test suite was measured.

6.1 Screen capture and OCR

The performance of screen capture and OCR was tested on a virtual machine, as described in section 5.1. Host system is running on Intel i7-1185G7 CPU. The virtual machine was running on QEMU with *virtio* graphics, which could be relevant for screen capture performance. The virtual machine resolution was set to 1280*800. *tesseract* 4.1.1 and *python* 3.9.14 were installed from RHEL repository. *ffmpeg* 5.1.2 was installed from EPEL. *opencv-python* 4.6.0.66 was installed from PyPI.

With this setup, the time for running each function was measured using `timeit` command from IPython. To measure the time of taking one screenshot, `Screen.screenshot` from GUI module was used. This function spends majority of the time waiting for *ffmpeg* to take a screenshot. Taking one screenshot took 780 ms on average with standard deviation of 7.05 ms.

To measure time of the image processing function `image_to_data` from GUI module was chosen. This function performs both image preprocessing (section 5.3.1) and OCR (section 5.3.2). The function was run on a screenshot of GDM of the resolution mentioned above, which should be fairly representative of the usage in tests. The resulting mean time was 310 ms with standard deviation of 96.7 ms.

Overall, taking one screenshot and converting it should not take more than a few seconds on any modern processor.

6.2 Running the tests in OpenStack

In production environment, the tests are going to run on Red Hat infrastructure as described in section 2.5.2. For this measurement, the tests were ran on Red Hat OpenStack.

Once the machine in the cloud was provisioned, the packages specified in the test metadata were installed. The provisioning and installation of the packages took 8 minutes, which is significantly higher compared to CLI-based smart card tests. The reason is the installation of the DNF group `Server with GUI`, which contains GNOME desktop environment and some basic applications.

Then the test script is run, which clones *SCAutolib* and tests based on *SCAutolib* from their git repositories. After that *SCAutolib* sets up the system for testing as described in section 2.5.3. The setup phase took 3 minutes and 50 seconds.

Each test case ran in a separate phase, with the exception of parametrized test cases. The run time of these phases ranged between 55 and 237 seconds. The whole suite of 8 test cases (4 of which were parametrized) took 15 minutes and 4 seconds to complete. All the test cases have passed, as can be seen in log in appendix C.

Overall, the installation of RHEL, installation of all the packages and running the tests took 23 minutes. Compared to manual testing, the time has been significantly reduced. If we assume that manual testing would take 4 working hours, automated tests are more than 10 times faster.

Chapter 7

Conclusion

The objective of this work was to automate the testing of smart card authentication in GUI in RHEL. In the research phase of the project, various existing approaches and tools have been examined. After evaluating all available options, it was concluded that none of them were suitable, hence a new GUI testing tool had to be developed. We have successfully designed and implemented the solution in form of module in *SCAutolib*. This module enables the tests to interact with GUI of the SUT (system under test). The GUI testing module offers a high-level API that enables the automation of tests in both the display manager and desktop environment. A set of test cases was written to cover basic authentication scenarios. These include logging in using GDM, and locking and unlocking the GNOME shell. Another target for testing is the lock on removal feature, which locks the screen automatically after the card is removed.

While not being the primary aim of this work, we have also implemented a script to run the new test cases in the Red Hat infrastructure. This way, the tests can run automatically, when a new build of RHEL has to be tested. As of now, the tests have been already deployed into production and they are used to test new builds of RHEL 9. The library was designed to work on future versions of RHEL. The main factor contributing to library's portability is its independence of the display server. The GUI tests are also compatible with RHEL 8, but the dependencies have to be installed manually. After these issues are solved RHEL 8 will be tested automatically as well.

In addition to GUI testing module, a module for checking the system logs was implemented. This module can be used to verify that a system log had been written. It is especially important when testing a failed login attempt, since this action needs to be traceable.

With the integration of GUI module, *SCAutolib* has gained the ability to test GUI programs. However, improvements could be made in other areas. For example, all the tests run on virtual machine and the tested cards are virtual. Running the tests on bare metal machine with real smart cards would be advantageous, since hardware cards still have to be tested manually. Another planned improvement of *SCAutolib* is the cleanup phase, which is insufficient in its current state. Proper cleanup would be beneficial when developing tests on a local virtual machine.

Bibliography

[1] Amazon Web Services, Inc. *What Is OCR (Optical Character Recognition)?* [cit. 2023-04-29]. Available at: https://aws.amazon.com/what-is/ocr/.

[2] ArchWiki contributors. *TigerVNC.* [cit. 2023-03-14]. Available at: https://wiki.archlinux.org/title/TigerVNC#Running_Xvnc_with_XDMCP_for_on_demand_sessions.

[3] Conservancy, S. F. *The Selenium Browser Automation Project.* [cit. 2023-04-26]. Available at: https://www.selenium.dev/documentation/.

[4] Dogtail contributors. *Dogtail README.* [cit. 2023-02-22]. Available at: https://gitlab.com/dogtail/dogtail.

[5] FFmpeg developers. *FFmpeg Devices Documentation.* [cit. 2023-02-16]. Available at: https://ffmpeg.org/ffmpeg-devices.html.

[6] Freedesktop.org. *Accessibility.* [cit. 2023-02-22]. Available at: https://www.freedesktop.org/wiki/Accessibility/.

[7] Freedesktop.org. *Add an API for taking screenshots and recording screencasts.* [cit. 2023-02-15]. Available at: https://gitlab.freedesktop.org/wayland/wayland/-/issues/32.

[8] Freedesktop.org. *AT-SPI2.* [cit. 2023-02-22]. Available at: https://www.freedesktop.org/wiki/Accessibility/AT-SPI2/.

[9] Freedesktop.org. *Project: p11-kit.* [cit. 2023-04-20]. Available at: https://p11-glue.github.io/p11-glue/p11-kit.html.

[10] Garousi, V., Afzal, W., Çaglar, A., Isik, I. B., Baydan, B. et al. Visual GUI testing in practice: An extended industrial case study. *CoRR.* 2020, abs/2005.09303. Available at: https://arxiv.org/abs/2005.09303.

[11] PKCS #11 Cryptographic Token Interface Base Specification Version 2.40. [cit. 2023-02-12]. Available at: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html.

[12] Graham, D., Fewster, M. and Rollins, J. *Foundations of Software Testing.* Cengage Learning EMEA, 2008. ISBN 9781844809899.

[13] Hocke, R. *SikuliX – the basics.* [cit. 2023-02-03]. Available at: https://sikulix-2014.readthedocs.io/en/latest/basicinfo.html.

[14] Høgsberg, K. *Wayland: The Wayland Protocol*. [cit. 2023-03-23]. Available at: https://wayland.freedesktop.org/docs/html/.

[15] ISO Central Secretary. Identification cards — Integrated circuit cards — Part 15: Cryptographic information. Geneva, CH: [b.n.]. 2016, ISO/IEC 7816:15. Available at: https://www.iso.org/standard/65250.html.

[16] ISO Central Secretary. Identification cards – Physical characteristics. Geneva, CH: [b.n.]. 2019, ISO/IEC 7810:2019. Available at: https://www.iso.org/standard/70483.html.

[17] Keyboard library contributors. *Keyboard library Documentation*. [cit. 2023-02-16]. Available at: https://github.com/boppreh/keyboard.

[18] Kiliēli, T. *Smart Card Overview*. [cit. 2023-02-09]. Available at: https://tldp.org/HOWTO/Smart-Card-HOWTO/index.html.

[19] Lauber, S. *An introduction to Pluggable Authentication Modules (PAM) in Linux*. [cit. 2023-04-20]. Available at: https://www.redhat.com/sysadmin/pluggable-authentication-modules-pam.

[20] Mouse library contributors. *Mouse library Documentation*. [cit. 2023-02-16]. Available at: https://github.com/boppreh/mouse/.

[21] Naccache, D. and M'Raihi, D. Cryptographic smart cards. *IEEE Micro*. 1996, vol. 16, no. 3, p. 14–24. DOI: 10.1109/40.502402. Available at: https://ieeexplore.ieee.org/document/502402.

[22] OpenCV contributors. *Template Matching*. [cit. 2023-02-15]. Available at: https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html.

[23] OpenCV team. *About*. [cit. 2023-02-16]. Available at: https://opencv.org/about/.

[24] openQA Team. *OpenQA Documentation*. [cit. 2023-01-27]. Available at: https://open.qa/docs/.

[25] OpenSC contributors. *Creating applications with smart card support*. [cit. 2023-02-09]. Available at: https://github.com/OpenSC/OpenSC/wiki/Creating-applications-with-smart-card-support.

[26] OpenSC contributors. *OpenSC documentation*. [cit. 2023-02-09]. Available at: https://github.com/OpenSC/OpenSC.

[27] Oracle. *Smart Card Overview*. [cit. 2023-02-09]. Available at: https://www.oracle.com/java/technologies/java-card/smartcards.html.

[28] pandas contributors. *Pandas documentation*. [cit. 2023-04-28]. Available at: https://pandas.pydata.org/docs/.

[29] Pierre, J. S. *The Linux Graphics Stack*. [cit. 2023-03-23]. Available at: https://blog.mecheye.net/2012/06/the-linux-graphics-stack/.

[30] PSCS Workgroup. *PC/SC Workgroup Compatible Products*. [cit. 2023-02-09]. Available at: https://pcscworkgroup.com/specifications/compatible-products/.

[31] PYTHON-UINPUT CONTRIBUTORS. *Python-uinput Documentation*. [cit. 2023-02-16]. Available at: https://github.com/tuomasjjrasanen/python-uinput.

[32] RED HAT, INC. *Configuring user authentication using authselect*. [cit. 2023-04-30]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html/configuring_authentication_and_authorization_in_rhel/configuring-user-authentication-using-authselect_configuring-authentication-and-authorization-in-rhel.

[33] RED HAT, INC. *Red Hat Enterprise Linux*. [cit. 2023-04-25]. Available at: https://www.redhat.com/rhel/.

[34] RED HAT, INC. *Release Notes for Red Hat Enterprise Linux 9.0*. [cit. 2023-02-15]. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html/9.0_release_notes/deprecated_functionality.

[35] RED HAT, INC. AND OTHERS. *Forbidden items*. [cit. 2023-02-28]. Available at: https://fedoraproject.org/wiki/Forbidden_items.

[36] ROCHETTE, T. *Tesseract TSV format*. [cit. 2023-02-22]. Available at: https://www.tomrochette.com/tesseract-tsv-format.

[37] ROUSSEAU, L. *PCSC lite project: Middleware to access a smart card using SCard API (PC/SC)*. [cit. 2023-02-08]. Available at: https://pcsclite.apdu.fr/.

[38] SMITH, R. *An Overview of the Tesseract OCR Engine*. [cit. 2023-04-28]. Available at: https://web.archive.org/web/20100928052954/http://tesseract-ocr.googlecode.com/svn/trunk/doc/tesseracticdar2007.pdf.

[39] SSSD CONTRIBUTORS. *SSSD Documantation*. [cit. 2023-04-20]. Available at: https://sssd.io/docs/introduction.html.

[40] TESSERACT CONTRIBUTORS. *Improving the quality of the output*. [cit. 2023-02-22]. Available at: https://tesseract-ocr.github.io/tessdoc/ImproveQuality.html.

[41] THE GNOME PROJECT. *Dconf*. [cit. 2023-02-28]. Available at: https://wiki.gnome.org/Projects/dconf.

[42] THE GNOME PROJECT. *GNOME Display Manager Reference Manual*. [cit. 2023-04-20]. Available at: https://help.gnome.org/admin/gdm/stable/.

[43] THE KERNEL DEVELOPMENT COMMUNITY. *Kernel Mode Setting (KMS)*. [cit. 2023-03-23]. Available at: https://dri.freedesktop.org/docs/drm/gpu/drm-kms.html.

[44] THE KERNEL DEVELOPMENT COMMUNITY. *Uinput module*. [cit. 2023-02-16]. Available at: https://kernel.org/doc/html/v4.12/input/uinput.html.

[45] TIGERVNC CONTRIBUTORS. *About TigerVNC*. [cit. 2023-03-14]. Available at: https://github.com/TigerVNC/tigervnc.

[46] X11VNC CONTRIBUTORS. *X11vnc: a VNC server for real X displays*. [cit. 2023-02-05]. Available at: https://github.com/LibVNC/x11vnc.

[47] X.Org Foundation. *X.Org*. [cit. 2023-04-28]. Available at: https://www.x.org/wiki/.

[48] Yadlouski, P. *SCAutolib documentation*. [cit. 2023-04-28]. Available at: https://scautolib.readthedocs.io.

[49] Yadlouski, P. *Automated Testing of Smart Cards*. Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/24161/.

Appendix A

Individual test cases with description

This appendix contains the list of implemented test cases with their description. The list is generated automatically by pytest framework from the docstrings. The command that generated this output is listed below.

```
pytest local-user-graphical-login.py local-user-lock-on-removal.py
 ↪  --collect-only
```

```
<Module Graphical/local-user-graphical-login.py>
  This module contains tests for logging into GUI using GDM.
  Most of the tests are parametrized to test both
  optional and required smart card in authselect.
  Lock-on-removal option is not set as it is irelevent for present tests.
  The tests within the module try logging in both using password and
  smart card with PIN. Both wrong password and wrong PIN are tested too.
  All tests depend on SCAutolib GUI module.

  If not stated otherwise tests in this module use virtual cards
  and share the following setup steps:
      1. Create local CA
      2. Create virtual smart card with certs signed by created CA
      3. Update /etc/sssd/sssd.conf so it contains following fields
          [sssd]
          debug_level = 9
          services = nss, pam,
          domains = shadowutils
          certificate_verification = no_ocsp
          [pam]
          debug_level = 9
          pam_cert_auth = True
          [domain/shadowutils]
          debug_level = 9
          id_provider = files
          [certmap/shadowutils/username]
          matchrule = <SUBJECT>.*CN=username.*
  <Function test_login_with_sc[local_user0-True]>
    Local user logs in with GDM, using smart card with correct PIN.
```

```
  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          OR
          authselect select sssd with-smartcard with-smartcard-required
      B. Start GDM
      C. Insert the card and type the correct PIN

  Expected result
      A. Configuration is updated
      B. GDM starts successfully
      C. User is asked for smartcard PIN and
          logged into GNOME desktop environment successfully
<Function test_login_with_sc[local_user0-False]>
  Local user logs in with GDM, using smart card with correct PIN.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          OR
          authselect select sssd with-smartcard with-smartcard-required
      B. Start GDM
      C. Insert the card and type the correct PIN

  Expected result
      A. Configuration is updated
      B. GDM starts successfully
      C. User is asked for smartcard PIN and
          logged into GNOME desktop environment successfully
<Function test_login_with_sc_wrong[local_user0-True]>
  Local user tries to log in with GDM, using smart card with wrong PIN.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          OR
          authselect select sssd with-smartcard with-smartcard-required
      B. Start GDM
      C. Insert the card and type an incorrect PIN

  Expected result
      A. Configuration is updated
      B. GDM starts successfully
      C. A message about incorrect PIN is displayed and user is not logged in.
<Function test_login_with_sc_wrong[local_user0-False]>
  Local user tries to log in with GDM, using smart card with wrong PIN.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          OR
          authselect select sssd with-smartcard with-smartcard-required
      B. Start GDM
      C. Insert the card and type an incorrect PIN

  Expected result
      A. Configuration is updated
      B. GDM starts successfully
      C. A message about incorrect PIN is displayed and user is not logged in.
```

```
  <Function test_login_password[local_user0]>
    Local user logs in with GDM using his password.

    Test steps
        A. Configure SSSD:
            authselect select sssd with-smartcard
        B. Start GDM
        C. Login as the user in GDM using password

    Expected result
        A. Configuration is updated
        B. GDM starts successfully
        C. User is successfully logged into GNOME desktop environment
  <Function test_login_password_wrong[local_user0]>
    Local user tries to log in with GDM using incorrect password.

    Test steps
        A. Configure SSSD:
            authselect select sssd with-smartcard
        B. Start GDM
        C. Try to log in using wrong password

    Expected result
        A. Configuration is updated
        B. GDM starts successfully
        C. A message about incorrect password is displayed
            and login is unsuccessful.
  <Function test_insert_card_prompt[local_user0]>
    Local user tries to log in with GDM before inserting card,
        with sc required.

        Test steps
        A. Configure SSSD:
            authselect select sssd with-smartcard
        B. Start GDM
        C. Insert the smart card
        D. Type the card's PIN

    Expected result
        A. Configuration is updated
        B. GDM starts successfully and "insert card" message is displayed
        C. GDM shows "insert PIN" prompt
        D. User is logged in successfully.
<Module Graphical/local-user-lock-on-removal.py>
  This module contains tests for lock on removal feature
  of GNOME desktop environment.
  Most of the tests are parametrized to test both
  optional and required smart card in authselect.
  All tests depend on SCAutolib GUI module.

  If not stated otherwise tests in this module use virtual cards
  and share the following setup steps:
      1. Create local CA
      2. Create virtual smart card with certs signed by created CA
      3. Update /etc/sssd/sssd.conf so it contains following fields
          [sssd]
          debug_level = 9
          services = nss, pam,
          domains = shadowutils
```

```
        certificate_verification = no_ocsp
        [pam]
        debug_level = 9
        pam_cert_auth = True
        [domain/shadowutils]
        debug_level = 9
        id_provider = files
        [certmap/shadowutils/username]
        matchrule = <SUBJECT>.*CN=username.*
<Function test_lock_on_removal[local_user0-True]>
  Local user removes the card while logged in to lock the screen.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          with-smartcard-lock-on-removal [with-smartcard-required]
      B. Start GDM and insert PIN to log in
      C. Remove the smart card
      D. Wake up the system by pressing enter key
          and unlock screen using PIN

  Expected result
      A. Configuration is updated
      B. GDM starts and user logs in successfully
      C. The system locks itself after the card is removed
      D. The system is unlocked
<Function test_lock_on_removal[local_user0-False]>
  Local user removes the card while logged in to lock the screen.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          with-smartcard-lock-on-removal [with-smartcard-required]
      B. Start GDM and insert PIN to log in
      C. Remove the smart card
      D. Wake up the system by pressing enter key
          and unlock screen using PIN

  Expected result
      A. Configuration is updated
      B. GDM starts and user logs in successfully
      C. The system locks itself after the card is removed
      D. The system is unlocked
<Function test_lock_on_removal_password[local_user0]>
  Local user inserts and removes the card while logged in with password.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          with-smartcard-lock-on-removal
      B. Start GDM and insert password to log in
      C. Insert the smart card
      D. Remove the smart card

  Expected result
      A. Configuration is updated
      B. GDM starts and user logs in successfully
      C. Nothing happens
      D. Nothing happens - system will not lock on card removal
```

```
<Function test_lockscreen_password[local_user0-True]>
  Local user unlocks screen using password, even if the smart card is
      inserted (after the password login). Screen unlocking requires the same
      method (PIN vs password) as was used for login.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          with-smartcard-lock-on-removal
      B. Start GDM and insert password to log in
      C. Insert the smart card
      D. Lock the screen manually
      E. Wake up and unlock the screen using password

  Expected result
      A. Configuration is updated
      B. GDM starts and user logs in successfully
      C. Nothing happens
      D. The screen is locked
      E. Screen is unlocked succesfully
<Function test_lockscreen_password[local_user0-False]>
  Local user unlocks screen using password, even if the smart card is
      inserted (after the password login). Screen unlocking requires the same
      method (PIN vs password) as was used for login.

  Test steps
      A. Configure SSSD:
          authselect select sssd with-smartcard
          with-smartcard-lock-on-removal
      B. Start GDM and insert password to log in
      C. Insert the smart card
      D. Lock the screen manually
      E. Wake up and unlock the screen using password

  Expected result
      A. Configuration is updated
      B. GDM starts and user logs in successfully
      C. Nothing happens
      D. The screen is locked
      E. Screen is unlocked succesfully
```

Appendix B

Files for running the test in Red Hat infracructure

Makefile

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#   Makefile of /CoreOS/smart-cards/Sanity/local-user-graphical
#   Description: Test basic authentication scenarios in GUI
#   Author: Ondrej Mach <omach@redhat.com>
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#   Copyright (c) 2021 Red Hat, Inc.
#
#   This program is free software: you can redistribute it and/or
#   modify it under the terms of the GNU General Public License as
#   published by the Free Software Foundation, either version 2 of
#   the License, or (at your option) any later version.
#
#   This program is distributed in the hope that it will be
#   useful, but WITHOUT ANY WARRANTY; without even the implied
#   warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
#   PURPOSE.  See the GNU General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with this program. If not, see http://www.gnu.org/licenses/.
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

export TEST=/CoreOS/smart-cards/Sanity/local-user-graphical
export TESTVERSION=1.0

BUILT_FILES=

FILES=$(METADATA) runtest.sh Makefile PURPOSE conf.json

.PHONY: all install download clean

run: $(FILES) build
	./runtest.sh

build: $(BUILT_FILES)
	test -x runtest.sh || chmod a+x runtest.sh
```

```
clean:
	rm -f *~ $(BUILT_FILES)


include /usr/share/rhts/lib/rhts-make.include

$(METADATA): Makefile
	@echo "Owner:           Ondrej Mach <omach@redhat.com>" > $(METADATA)
	@echo "Name:            $(TEST)" >> $(METADATA)
	@echo "TestVersion:     $(TESTVERSION)" >> $(METADATA)
	@echo "Path:            $(TEST_DIR)" >> $(METADATA)
	@echo "Description:     Test basic authentication scenarios in GUI" >>
	↪ $(METADATA)
	@echo "Type:            Sanity" >> $(METADATA)
	@echo "TestTime:        30m" >> $(METADATA)
	@echo "RunFor:          smart-cards" >> $(METADATA)
	@echo "Requires:        git sssd httpd softhsm opensc openssl gnutls-utils
	↪ nss-tools nss-pam-ldapd python3-pip gdm sssd-tools pcsc-lite-ccid
	↪ pcsc-lite pcsc-lite-libs authselect ffmpeg-free tesseract gcc" >>
	↪ $(METADATA)
	@echo "RhtsRequires:    library(distribution/epel)" >> $(METADATA)
	@echo "Priority:        Normal" >> $(METADATA)
	@echo "License:         GPLv2+" >> $(METADATA)
	@echo "Confidential:    no" >> $(METADATA)
	@echo "Destructive:     no" >> $(METADATA)
	@echo "Releases:        -RHEL4 -RHELClient5 -RHELServer5 -RHEL6 -RHEL7" >>
	↪ $(METADATA)

	rhts-lint $(METADATA)
```

runtest.sh

```
#!/bin/bash
# vim: dict+=/usr/share/beakerlib/dictionary.vim cpt=.,w,b,u,t,i,k
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#   runtest.sh of /CoreOS/smart-cards/Sanity/local-user-graphical
#   Description: Test basic authentication scenarios in GUI
#   Author: Ondrej Mach <omach@redhat.com>
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
#   Copyright (c) 2021 Red Hat, Inc.
#
#   This program is free software: you can redistribute it and/or
#   modify it under the terms of the GNU General Public License as
#   published by the Free Software Foundation, either version 2 of
#   the License, or (at your option) any later version.
#
#   This program is distributed in the hope that it will be
#   useful, but WITHOUT ANY WARRANTY; without even the implied
#   warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
#   PURPOSE.  See the GNU General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with this program. If not, see http://www.gnu.org/licenses/.
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Include Beaker environment
```

```
. /usr/share/beakerlib/beakerlib.sh || exit 1

PACKAGES="sssd httpd softhsm opensc openssl gnutls-utils nss-tools python3-pip
↪ sssd-tools pcsc-lite-ccid pcsc-lite pcsc-lite-libs authselect ffmpeg-free
↪ tesseract gcc"

rlJournalStart
    rlPhaseStartSetup
        rlRun "rlImport 'distribution/epel'"
        rlRun "epel yum -y install ffmpeg-free"
        for package in $PACKAGES; do
            rlAssertRpm $package
        done
        rlRun "TmpDir=\$(mktemp -d)" 0 "Creating tmp directory"
        rlRun "cp ./* $TmpDir"
        rlRun "pushd $TmpDir"
        rlRun "CONF=$TmpDir/conf.json"

        #Backup files
        rlRun "rlFileBackup --missing-ok --namespace sc_test /etc/sssd/sssd.conf"
        rlRun "rlFileBackup --clean --missing-ok --namespace sc_test /var/lib/sss/"
        rlRun "rlFileBackup --clean --missing-ok --namespace sc_test
        ↪ /usr/share/p11-kit/modules/opensc.module"
        rlRun "rlFileBackup --clean --missing-ok --namespace sc_test /etc/dconf/db/"

        # Install library and clone tests
        rlRun "python3 -m pip install --upgrade pip"
        rlRun "python3 -m pip install SCAutolib[graphical]"
        rlRun "git clone -b V2-adoptation
        ↪ https://github.com/redhat-qe-security/SC-tests.git"

        rlRun "pip3 install --upgrade -I -r $TmpDir/SC-tests/requirements.txt"
        rlRun "scauto --conf $CONF --verbose DEBUG --force prepare --graphical
        ↪ --install-missing"
    rlPhaseEnd

    rlPhaseStartTest "Local user with SC with and without sc-required"
        rlRun "pytest
        ↪ SC-tests/Graphical/local-user-graphical-login.py::test_login_with_sc -sv"
    rlPhaseEnd

    rlPhaseStartTest "Local user with SC (wrong PIN) with and without sc-required"
        rlRun "pytest SC-tests/Graphical/local-user-graphical-login.py::test_login_wi
        ↪ th_sc_wrong
        ↪ -sv"
    rlPhaseEnd

    rlPhaseStartTest "Local user with password, sc not required"
        rlRun "pytest
        ↪ SC-tests/Graphical/local-user-graphical-login.py::test_login_password -sv"
    rlPhaseEnd

    rlPhaseStartTest "Local user with wrong password, sc not required"
        rlRun "pytest SC-tests/Graphical/local-user-graphical-login.py::test_login_pa
        ↪ ssword_wrong
        ↪ -sv"
    rlPhaseEnd

    rlPhaseStartTest "Test insert prompt, when sc is not inserted, but required"
```

```
        rlRun "pytest
        ↪ SC-tests/Graphical/local-user-graphical-login.py::test_insert_card_prompt
        ↪ -sv"
    rlPhaseEnd

    rlPhaseStartTest "Lock on removal with and without sc-required"
        rlRun "pytest
        ↪ SC-tests/Graphical/local-user-lock-on-removal.py::test_lock_on_removal
        ↪ -sv"
    rlPhaseEnd

    rlPhaseStartTest "Lock on removal when logged in with parssword"
        rlRun "pytest SC-tests/Graphical/local-user-lock-on-removal.py::test_lock_on_
        ↪ removal_password
        ↪ -sv"
    rlPhaseEnd

    rlPhaseStartTest "Test lockscreen when logged in with password"
        rlRun "pytest SC-tests/Graphical/local-user-lock-on-removal.py::test_lockscre
        ↪ en_password
        ↪ -sv"
    rlPhaseEnd

    rlPhaseStartCleanup
        rlRun "rlFileRestore --namespace sc_test"
        rlRun "popd"
        rlRun "pip uninstall SCAutolib -y"
        if grep base-user /etc/passwd; then
            rlRun "userdel -f base-user"
        fi
        rlRun "rm -r $TmpDir" 0 "Removing tmp directory"
    rlPhaseEnd
rlJournalPrintText
rlJournalEnd
```

conf.json

```
{
  "root_passwd": "redhat",
  "ca": {
    "local_ca": {}
  },
  "users": [
    {
      "name": "local-user",
      "passwd": "654321",
      "pin": "123456",
      "card_dir": "/root/local-user",
      "card_type": "virtual",
      "local": true
    }
  ]
}
```

Appendix C

Log from successful test run

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   TEST PROTOCOL
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

    Test run ID   : 9235001
    Package       : smart-cards
    beakerlib RPM : beakerlib-1.29.3-1.el9.noarch
    bl-redhat RPM : beakerlib-redhat-1-33.el9.noarch
    Test name     : /CoreOS/smart-cards/Sanity/local-user-gui
    Test version  : 1.0
    Test started  : 2023-04-17 11:04:23 EDT
    Test finished : 2023-04-17 11:23:17 EDT (still running)
    Test duration : 1134 seconds
    Distro        : Red Hat Enterprise Linux release 9.2 (Plow)
    Hostname      : ci-vm-10-0-139-159.hosted.upshift.rdu2.redhat.com
    Architecture  : x86_64
    CPUs          : 1 x Intel Xeon Processor (Icelake)
    RAM size      : 1776 MB
    HDD size      : 19.98 GB

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Test description
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

PURPOSE of /CoreOS/smart-cards/Sanity/local-user-graphical
Author: Ondrej Mach <omach@redhat.com>

Description: Test smart card authentication in GUI for local user


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Setup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:04:23 ] :: [   PASS   ] :: Checking for the presence of sssd rpm
:: [ 11:04:23 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:23 ] :: [   LOG    ] ::   sssd-2.8.2-2.el9.x86_64
:: [ 11:04:23 ] :: [   PASS   ] :: Checking for the presence of httpd rpm
:: [ 11:04:23 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:23 ] :: [   LOG    ] ::   httpd-2.4.53-11.el9_2.4.x86_64
:: [ 11:04:23 ] :: [   PASS   ] :: Checking for the presence of softhsm rpm
:: [ 11:04:23 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:23 ] :: [   LOG    ] ::   softhsm-2.6.1-7.el9.2.x86_64
```

```
:: [ 11:04:23 ] :: [   PASS   ] :: Checking for the presence of opensc rpm
:: [ 11:04:23 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:23 ] :: [   LOG    ] ::   opensc-0.22.0-2.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of openssl rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   openssl-3.0.7-6.el9_2.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of gnutls-utils rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   gnutls-utils-3.7.6-20.el9_2.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of nss-tools rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   nss-tools-3.79.0-18.el9_1.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of python3-pip rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   python3-pip-21.2.3-6.el9.noarch
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of sssd-tools rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   sssd-tools-2.8.2-2.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of pcsc-lite-ccid rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   pcsc-lite-ccid-1.4.36-1.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of pcsc-lite rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   pcsc-lite-1.9.4-1.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of pcsc-lite-libs rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   pcsc-lite-libs-1.9.4-1.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of authselect rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   authselect-1.2.6-1.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of ffmpeg-free rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   ffmpeg-free-5.1.3-1.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of tesseract rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   tesseract-4.1.1-7.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Checking for the presence of gcc rpm
:: [ 11:04:24 ] :: [   LOG    ] :: Package versions:
:: [ 11:04:24 ] :: [   LOG    ] ::   gcc-11.3.1-4.3.el9.x86_64
:: [ 11:04:24 ] :: [   PASS   ] :: Creating tmp directory (Expected 0, got 0)
:: [ 11:04:24 ] :: [   PASS   ] :: Command 'cp ./* /tmp/tmp.aozuggS68t' (Expected 0, got 0)
:: [ 11:04:24 ] :: [   PASS   ] :: Command 'pushd /tmp/tmp.aozuggS68t' (Expected 0, got 0)
:: [ 11:04:24 ] :: [   PASS   ] :: Command 'CONF=/tmp/tmp.aozuggS68t/conf.json' (Expected
↪  0, got 0)
:: [ 11:04:24 ] :: [   INFO   ] :: using '/var/tmp/beakerlib-9235001/backup-sc_test' as
↪  backup destination
:: [ 11:04:24 ] :: [   PASS   ] :: Command 'rlFileBackup --missing-ok --namespace sc_test
↪  /etc/sssd/sssd.conf' (Expected 0, got 0)
:: [ 11:04:24 ] :: [   INFO   ] :: using '/var/tmp/beakerlib-9235001/backup-sc_test' as
↪  backup destination
:: [ 11:04:24 ] :: [   PASS   ] :: Command 'rlFileBackup --clean --missing-ok --namespace
↪  sc_test /var/lib/sss/' (Expected 0, got 0)
:: [ 11:04:24 ] :: [   INFO   ] :: using '/var/tmp/beakerlib-9235001/backup-sc_test' as
↪  backup destination
:: [ 11:04:24 ] :: [   PASS   ] :: Command 'rlFileBackup --clean --missing-ok --namespace
↪  sc_test /usr/share/p11-kit/modules/opensc.module' (Expected 0, got 0)
:: [ 11:04:24 ] :: [   INFO   ] :: using '/var/tmp/beakerlib-9235001/backup-sc_test' as
↪  backup destination
```

```
:: [ 11:04:24 ] :: [   PASS   ] :: Command 'rlFileBackup --clean --missing-ok --namespace
↪  sc_test /etc/dconf/db/' (Expected 0, got 0)
:: [ 11:04:27 ] :: [   PASS   ] :: Command 'python3 -m pip install --upgrade pip' (Expected
↪  0, got 0)
:: [ 11:04:27 ] :: [   PASS   ] :: Command 'git clone -b V2-graphical
↪  https://github.com/ondrej-mach/SCAutolib.git' (Expected 0, got 0)
:: [ 11:04:46 ] :: [   PASS   ] :: Command 'python3 -m pip install
↪  /tmp/tmp.aozuggS68t/SCAutolib[graphical]' (Expected 0, got 0)
:: [ 11:04:46 ] :: [   PASS   ] :: Command 'git clone -b graphical
↪  https://github.com/ondrej-mach/SC-tests.git' (Expected 0, got 0)
:: [ 11:04:47 ] :: [   PASS   ] :: Command 'pip3 install --upgrade -I -r
↪  /tmp/tmp.aozuggS68t/SC-tests/requirements.txt' (Expected 0, got 0)
:: [ 11:08:13 ] :: [   PASS   ] :: Command 'scauto --conf /tmp/tmp.aozuggS68t/conf.json
↪  --verbose DEBUG --force prepare --graphical --install-missing' (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 230s
::   Assertions: 30 good, 0 bad
::   RESULT: PASS (Setup)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Local user with SC with and without sc-required
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:09:49 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-graphical-login.py::test_login_with_sc -sv' (Expected 0,
↪  got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 96s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Local user with SC with and without sc-required)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Local user with SC (wrong PIN) with and without sc-required
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:11:24 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-graphical-login.py::test_login_with_sc_wrong -sv'
↪  (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 95s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Local user with SC (wrong PIN) with and without sc-required)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Local user with password, sc not required
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:12:19 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-graphical-login.py::test_login_password -sv' (Expected 0,
↪  got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 55s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Local user with password, sc not required)
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Local user with wrong password, sc not required
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:13:15 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-graphical-login.py::test_login_password_wrong -sv'
↪  (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 56s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Local user with wrong password, sc not required)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Test insert prompt, when sc is not inserted, but required
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:14:13 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-graphical-login.py::test_insert_card_prompt -sv'
↪  (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 58s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Test insert prompt, when sc is not inserted, but required)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Lock on removal with and without sc-required
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:17:58 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-lock-on-removal.py::test_lock_on_removal -sv' (Expected
↪  0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 225s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Lock on removal with and without sc-required)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Lock on removal when logged in with parssword
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:19:19 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-lock-on-removal.py::test_lock_on_removal_password -sv'
↪  (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 81s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Lock on removal when logged in with parssword)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Test lockscreen when logged in with password
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:23:16 ] :: [   PASS   ] :: Command 'pytest
↪  SC-tests/Graphical/local-user-lock-on-removal.py::test_lockscreen_password -sv'
↪  (Expected 0, got 0)
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 237s
::   Assertions: 1 good, 0 bad
::   RESULT: PASS (Test lockscreen when logged in with password)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Cleanup
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:23:16 ] :: [   PASS   ] :: Command 'rlFileRestore --namespace sc_test' (Expected 0,
↪  got 0)
:: [ 11:23:16 ] :: [   PASS   ] :: Command 'popd' (Expected 0, got 0)
:: [ 11:23:17 ] :: [   PASS   ] :: Command 'pip uninstall SCAutolib -y' (Expected 0, got 0)
:: [ 11:23:17 ] :: [   PASS   ] :: Command 'userdel base-user' (Expected 0, got 0)
:: [ 11:23:17 ] :: [   PASS   ] :: Removing tmp directory (Expected 0, got 0)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 1s
::   Assertions: 5 good, 0 bad
::   RESULT: PASS (Cleanup)


::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   /CoreOS/smart-cards/Sanity/local-user-gui
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [ 11:23:17 ] :: [   LOG    ] :: Phases fingerprint:  4EgrNTvG
:: [ 11:23:17 ] :: [   LOG    ] :: Asserts fingerprint: OAO4hTvI
:: [ 11:23:17 ] :: [   LOG    ] :: File '/var/tmp/beakerlib-9235001/journal.xml' stored
↪  here: /var/tmp/BEAKERLIB_9235001_STORED_journal.xml
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::   Duration: 1134s
::   Phases: 10 good, 0 bad
::   OVERALL RESULT: PASS (/CoreOS/smart-cards/Sanity/local-user-gui)
```