March 2023

TECHNIQUES TO REDUCE THE RISK OF MALICIOUS OPERATIONS BEING PERFORMED DURING REMOTE CONTROL OF NETWORK DEVICES

Yi Ding

Humphrey Hanfeng Cai

Roman Volkov

Samantha Wang

TECHNIQUES TO REDUCE THE RISK OF MALICIOUS OPERATIONS BEING PERFORMED DURING REMOTE CONTROL OF NETWORK DEVICES

AUTHORS:
Yi Ding
Humphrey Hanfeng Cai
Geetu
Roman Volkov
Samantha Wang

ABSTRACT

Remote logins to network devices, such as during remote support sessions, can potentially introduce the risk of harm to network devices through human errors that could be triggered by technical support personnel during such remote logins/support sessions. Presented herein are techniques that can be implemented to fundamentally reduce the risk of human error that may occur during remote control of network equipment.

DETAILED DESCRIPTION

There are often risks of human error (maloperation) when technical support personnel provide remote support to a person, customer, etc. via a remote tool (e.g., video conferencing tools, remote login/debugging tools, etc.), such as during the live debugging of a customer's network devices. For example, in such scenarios there are risks that a command may be input within an incorrect console window when technical support personnel are operating multiple console window via a remote tool while a customer is not paying attention to the technical support personnel's actions. Some maloperations can cause business impacts, such as reloading, restarting, clearing tables, and/ or triggering command-line interface (CLI) bugs.

Although human error may not be prevented during remote control sessions, it would be advantageous if the occurrence of maloperations could be reduced or even eliminated through a technical approach. On the other hand, it would be useful if a customer could be provided with the ability to review/permit/deny certain actions/commands that may be sent from a remote control terminal to network devices in order to protect interests of the customer.

Presented herein are techniques that can be provided in order to prevent or mitigate risks of maloperations during remote support sessions/control of a customer's network device(s). In particular, techniques presented herein may involve providing logic for any remote support tool that provides for the ability to audit remote command inputs to confirm that the inputs will not trigger unexpected actions, such as triggering known bugs, causing maloperations to be performed on customer devices, etc.

In accordance with techniques of this proposal, auditing of a send-command operation can be divided into three steps: (1) front-end typing → (2) Enter → (3) Send to Back-end. In particular, a "sentry" plug-in or logic can be integrated into the operational flow between steps (2) and (3) to facilitate audit protection for remote commands. The feature can be enabled/disabled based on user preferences/settings.

When the feature is enabled, the command contents will be required to execute the process flow as shown below in Figure 1.

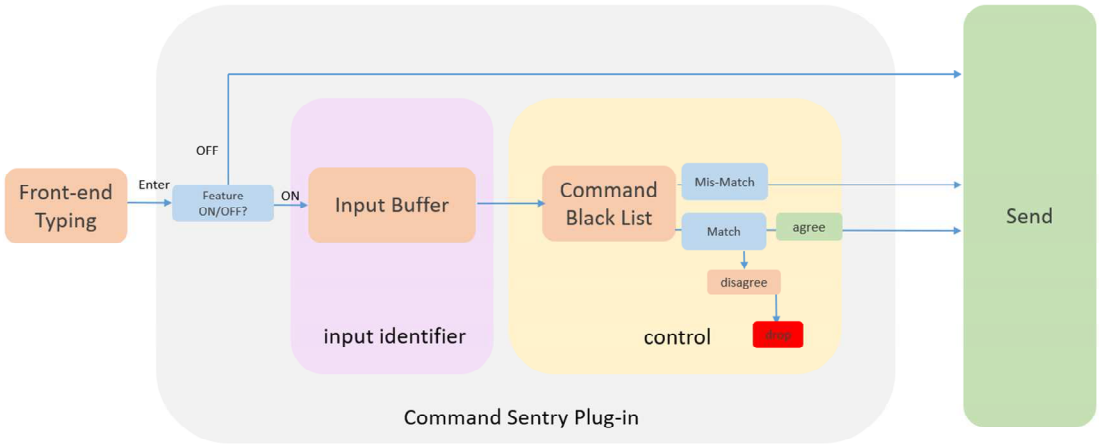

Figure 1: Example Command Sentry Process Flow

With regard to Figure 1, at the "Identify" step for the process flow, the keyboard's input can be held/cached while an action result is determined by the "control" component. The control component can match the command against a blacklist database such that, if an entry is matched, then permission can be requested from the remote user (customer) to determine if they will allow or not allow the command to be executed. However, in no entry is matched, then the command can be output directly to the remote user's device (as

shown via the "Send" component, in which the final command can be sent to the back-end device).

The blacklist database can be created through a variety of techniques. For example, in some instances users can create a customized blacklist. In another example, a provider of the command sentry plug-in can automatically populate the blacklist database with risky commands. For instance, in some cases the blacklist database can be a table that stores sensitive commands (e.g., known commands that can cause memory leaks, crashes, high CPU usage, etc.) that can be identified via a known defect database.

In one example, as shown in Figure 2, known defect checker logic can be utilized to query/obtain commands from a known defect database and update/cache the commands to the blacklist using various Application Programming Interface (API) operations such that the blacklist can be continually updated with new/known sensitive commands.

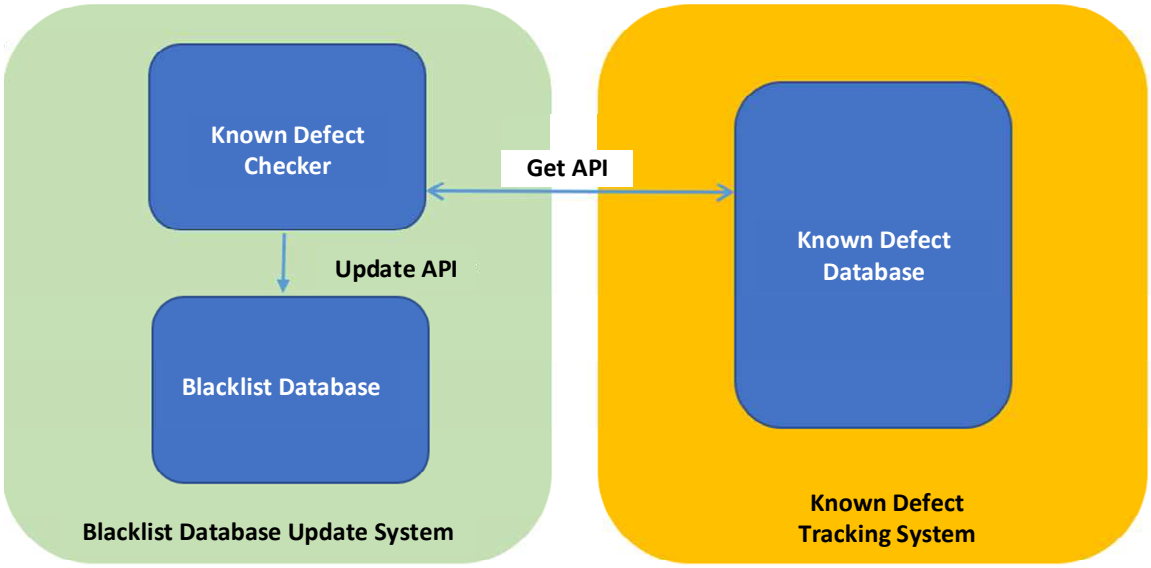

Figure 2: Automatic Blacklist Population

With regard to automatic population of the blacklist database, as shown in Figure 2, it can be useful to automatically filter risky commands to the blacklist. For example, with regard to the known defect database as shown in Figure 2, one or more tags can be added to each entry of the database that indicates whether the entry is a "show" CLI-triggered issue. If an entry is a show CLI-triggered issue, the entire "show" CLI can be input into the tag or it can be labeled as "n/a".

Using the new metadata tag “show cli triggered”, the sentry plug-in can determine whether a current expected CLI may be harmful to remote equipment. Based API interactions with the known defect database, accurate reports can be obtained to determine whether CLIs are safe to execute or should be added to the blacklist database.

Further, in some instances, known defect identifiers can be used by technical support personnel to double-check whether current a platform and/or software of a remote device may be affected by a given CLI. Additionally, in some implementations, the known defect checker logic and be configured to iterate over the known defect database on a daily basis (or over some other configurable period of time) in order to ensure that the blacklist database represents the most up-to-date list of blacklisted commands.

In some instances, incomplete commands can be input by technical support personnel, which should also be checked against the blacklist database. In order to parse incomplete commands in relation to the blacklist database, blacklisted commands can be split and stored in the database.

For example, a blacklisted command can be stored in the blacklist database along with various other information/elements that can be used to identify a given command, such as its known defect identifier, the complete command string, the length of the command string, update date/time, bug title, and a list of one or more split command word that may be considered as an incomplete command for a given complete command (e.g., split command word[0] = 'XXXXX', split command word[1] = 'YYYYY', etc.).

Since the typical command format involves each word being separated by a space, the following logic can be utilized to determine whether a given command is abbreviated/incomplete and is included in the blacklist database, as follows:

1. Split input command by space and assign into a list 'A';
2. Filter entries with list A's length value (e.g., database entries should have command length element);
3. Compare with the entries one by one as filtered by step 2.
4. For each entry, compare with each split command word[0] and listA[0], word[1] and listA[1], and so on, in order to determine whether listA[x] is included and as start letters of word[x]. If NOT included, then return as 'False'.

5. If there is entry that returns as 'True', then return as "Blacklist Matched'.

Figure 3, below, illustrates an example Python program that can be utilized to check for incomplete word matches, in one instance.

```
def __cmp_cmd():
    _input = "sh int bri"
    _bl_cmd = "show interface brief"
    _input_cmd_split = _input.split()
    _bl_cmd_split = _bl_cmd.split()
    _cmd_len = len(_input_cmd_split)

    if len(_input_cmd_split) == len(_bl_cmd_split):
        i = 0
        while i < _cmd_len:
            p = _bl_cmd_split[i].find(_input_cmd_split[i])
            if p == 0:
                i = i + 1
            else:
                return False
        return True
    else:
        return False
```

Figure 3: Example Python Program to Check for Incomplete Commands

In summary, techniques herein provide for the ability to prevent or mitigate the risk of maloperations during a remote support session by checking CLI commands, both complete and incomplete, that may bey input by remote support personnel against a database of blacklisted commands. Further, in some instances, end users/customers can be responsible for allowing or not allowing certain commands to be executed. Such techniques can be implemented in order to avoid known defects being triggered in an end user's/customer's network and, thus, can fundamentally reduce the risk of human error during remote control of network equipment.