Poster: PatchGen: Towards Automated Patch Detection and Generation for 1-Day Vulnerabilities

Tianyue Luo
General Department
Institute of Software, The Chinese Academy of Sciences
tianyue@iscas.ac.cn

Chen Ni
General Department
Institute of Software, The Chinese Academy of Sciences
nichen@nfs.iscas.ac.cn

Qing Han
General Department
Institute of Software, The Chinese Academy of Sciences
hanqing@iscas.ac.cn

Mutian Yang
General Department
Institute of Software, The Chinese Academy of Sciences
mutian@nfs.iscas.ac.cn

Jingzheng Wu
General Department, State Key Laboratory of Computer Sciences
Institute of Software, The Chinese Academy of Sciences
jingzheng08@iscas.ac.cn

Yanjun Wu
General Department, State Key Laboratory of Computer Sciences
Institute of Software, The Chinese Academy of Sciences
yanjun@iscas.ac.cn

ABSTRACT

A large fraction of source code in open-source systems such as Linux contain 1-day vulnerabilities. The command "patch" is used to apply the patches to source codes, and returns feedback information automatically. Unfortunately, this operation is not always successful when patching directly, and two typical error scenarios may occur as follows. 1. The patch may be applied in wrong place, meaning the fix location should be adjusted in patch. 2. The patch may be applied repeatedly, meaning a verification should be executed before applying. To resolve the above scenarios, we propose PatchGen, a new system to quickly detect and generate patches for 1-day vulnerabilities in OS distributions. Comparing with the previous works on 1-day vulnerabilities detection, PatchGen is able to solve the above two error scenarios and use a quick, syntax-based approach that scales to OS distribution-sized code base no matter the code written in what types of language. We implement the PatchGen prototype, and evaluate it by checking all codes from packages in Ubuntu Maverick/Oneiric, all SourceForge C and C++ projects, and the Linux kernel source. Specifically, it takes less than 10 minutes for PatcheGen to detect 175 1-day vulnerabilities and generate 140 patches for Linux Kernel. All of the results have been manually confirmed and tested in the real systems.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
ACM 978-1-4503-3832-5/15/10.
http://dx.doi.org/10.1145/2810103.2810122.

Keywords

Patch; 1-Day Vulnerabilities; Automated Generation; Scalability

1. INTRODUCTION

In open-source systems such as Linux, developers use patch to fix bugs. Although the "patch" command can apply a patch to source code and return feedback information automatically, the execution may not be successfully[3, 2]. That is, the result is not necessarily true. Here are two error scenarios when patching directly.

1. The source code need to be changed, but it fixes a wrong place while patching directly. That means the patch should be adjusted before patching [1].

2. The source file has been fixed, but if patched again, the source code can also be changed. That means the patch whether has been applied must be detected.

The situations described above is common/can't be ignored, and we should keep in mind while patching source code. But if there are a lot of patches should be apply at the same time, the examination will be a tedious work. This will be more obverse when the source code is different from the version the patch originally pointed at, especially for the costumed system based on open-source systems[5, 4]. Above all, batch patching a software code poses two main challenges: First, whether the source code has the questions as the patches described must be determined automatically. Second, typical checks must be performed to make sure the patches can be patched correctly. Moreover, the result must be easily comprehended and presented to user friendly.

2. DESIGN OF PATCHGEN

PatchGen has two major functionalities: detecting code segments that are same with patches in database, then identifying whether the patches should be patched and adjusting them. It requires no modification to the source code of software being analyzed. The following subsections will describe the design for each functionality. Our method implements a four-step procedure to solve these problems. This procedure

is motivated by the design space goals of: (1) focusing on unpatched code's positions, (2) scaling to large and diverse customized code bases such as a new kernel based Ubuntu, (3) minimizing false detection. The core of the PatchGen system is illustrated in Figure 1 and the steps are outlined in the following:

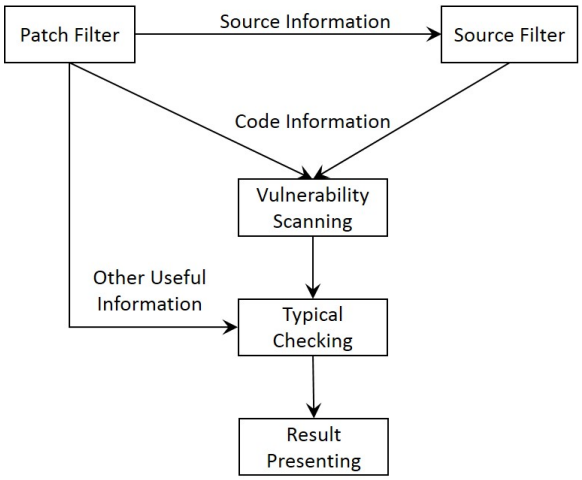


Figure 1: The Overview Design of PatchGen

1. Patch extraction. PatchGen extracts the related source files'information, original code snippets, patched code snippets and position identifiers firstly. As comments are not considered normal statements in PatchGen, and are thereby filtered at this step. Farther more, we normalizes each code snippet by removing the redundant whitespace, curly braces and non ASCII characters. 2. Source file grouping. According to the information extracted in the first step, PatchGen divides the source files and patches into groups, which is consisted of a set of patches and a source file related to them. For each source file related to the patch, PatchGen performs the same normalize operation. 3. Vulnerability scanning. PatchGen executes this step by group using a Sliding Window Algorithm, the length n of the window is decided by the normalized patch file. Then the source file is compared with patch segments, and possible unpatched code positions are marked at the same time. 4. Pruning False Positives. As matched code snippets are not necessarily the vulnerability, PatchGen performs a filter operation on the identified unpatched codes, such that the false detections can be reduced. 5. Report generation. In order to show the detail of the detection, a report is generated, which can also help to accelerate the manual auditing of source code.

2.1 Patch Extraction

Patch is in the form of unified diffs, which always consists of one or more sequence of diff hunks that contain the line difference in the file. The diff hunk starts with two almost same lines as header, the only difference is that the original file is prefixed by "—" but the new file is prefixed by "+++". The unchanged, contextual lines are prefixed by a space character, added source code lines are prefixed by a plus sign "+", and deletions are prefixed by a minus sign "-".

PatchGen extracts the original code segment and the patched code segment from given patch file respectively. To extract the original code segment, we reserves the lines started with the sign"-", and deletes the ones started with sign "+". And we do the reverse operation to obtain the patched code segment. To remove the influence caused by code nuance, PatchGen normalizes each patch file: removes typical language comments, all non-ASCII characters and redundant whitespace. Position information after identifier "@@" is also extracted for filter operation described in step4.

2.2 Source File Grouping

In most cases, we can start vulnerability scanning after extracting useful information from patches, which is similar with the process of code clones detection. There are many ways to find code clones in amount of source filesčňmost of which search the whole source files to find all code clones and then check if they are patched. This process always consumes too much time. In consideration that we only want to identify whether the patch is patched, we can use the information included in the format context. The two-line header before diff hunks indicates the source file the patch influenced, so that we can just focus in this file while doing bug finding. Aware of there are many different patches influence the same source file, PatchGen groups these patches according to the context format before diff hunk to further reduce the scanning time. For each source file related to the patches, PatchGen performs the same normalize operation at the same time.

2.3 Unpatched Code Checking

Through the two steps above, we get a sequence of groups, which is consisted of a set of patch blocks and a source file related to them. Each patch block contains the influenced source filename, the original code segment, the patched code segment and position information. PatchGen uses a string-based techniques to detect whether the code is patched. By dividing the normalized source file in to strings based on the original code snippet got from the patch block, we preforms an exact match on each window. The compared window slips backward until they matched, and the code segments detected are potential defect code and need future analysis.

2.4 Pruning False Positives

Research shows that code clones is very common in large-scale software, the matched code segments in step3 may not be the vulnerability we expected, although we have narrowed down the detection to the specified file. Just as the problem described in section2, which indicates that a farther process should be done to reduce the false positives. The range information before each diff hunk, which is surrounded by double-at signs "@", tells the changed position of the source file. However, due to version differences, this information is not always accurate. PatchGen use another position information after sign"@@", which tells the heading of the section or function that the hunk is part of. We check the identifier after"@@" for each matched diff hunk, then do the subprocess based on the rules described in table1.

If the identifier after "@@" can be found in source file and the position of matched code segment is behind the identifier at the same time, that means the reverse patch detection is needed. PatchGen uses the patched code snippet extracted in step1 to do a farther comparison with the source file. Different from the first time detection, we just compare the code after the position where the identifier first appeared. The origin patch block will be considered as a right one while

Table 1: Rules for Subprocess in PatchGen

Situation of identifier after "@@"	Rule description
Exist & Can't be found in source file	The corresponding matched code segment is false positive and can be filtered out
Exist & The position of matched code segmentis before the identifier	The corresponding matched code segment is false positive and can be filtered out
Exit & The position of matched code segment is behind the identifier	Go to reverse patch detection. That is, PatchGen uses patched code snippet to compare with source code. This time we only compare with part of source file, starting from line number where "@@" locates. No matching results means that this patch block is indeed a right one. Otherwise, go to manual auditing to confirm whether this match is false positive.
Empty	Go to manual auditing.

none is found this time, otherwise, we should do a manual auditing to confirm whether the first match is false positive.

2.5 Report Generation

Although PatchGen has given a group of adjusted patches which should be patched on the system and made a filter operation on the result, a report is still needed to illustrate the details. A report contains patch blocks, the corresponding detected locations, and the contextual code. This can assist analysts in deciding whether use the patches, even in finding and adjusting a patch for potential vulnerability.

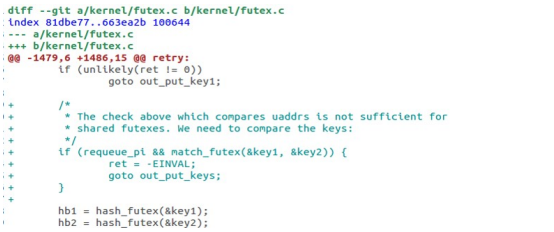
3. EVALUATION

We conducted a 10-versions of Linux Kernel experiment to test PatchGen.The accumulated detected 1187 1-day vulnerabilities and generate 1187 correspondence patches.In the results, PatchGen has successfully solved the two error scenarios in the previously mentioned. Figure2 shows a hunk in CVE-2014-3153 and parts of its corresponding source code in the file /kernel/futex.c of Linux Kernel-3.13.0-24.47. There are two places can be patched: segment 1 and segment 2. Wherein segment 1 doesn't need to patch . PatchGen based on an analysis to fix segment 2 and it can just change this segment. However, if the execution of "patch" directly ,according to rules, segment 1 is fixed and this patch fixes a wrong place.

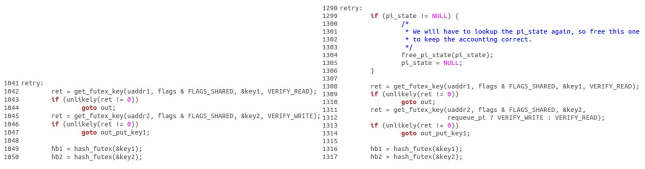
A hunk in CVE-2011-2022 and parts of its corresponding source code in the file /kernel/futex.c of Linux Kernel-3.13.0-24.47. The execution of "patch" command will patch code segment from line 1087 to line 1091 showed in subgraph b, and return a successful result. In fact, PatchGen generate the patch should fix at the code segment from line 1140 to line 1150.

4. CONCLUSION

We present PatchGen in this poster, which is a system for quickly generate patch of 1-day vulnerable code. PatchGen



(a) A hunk in CVE-2014-3153



(b) Matched segment 1 and segment 2

Figure 2: A hunk in CVE-2014-5153 and parts of matched code

represents that uses a quick, syntax-based approach that scales to OS distribution-sized code base that include code written in many different languages. At the same time, PatchGen has the ability to handle real code, and minimizing false detection of 1-day vulnerabilities. PatchGen generate 1187 patches, which likely fix real vulnerabilities, by analyzing more than 2 billion lines of code. We believe PatchGen can be realistic solution for developers to enhance the security of their code in the future work.

5. ACKNOWLEDGMENTS

This work is supported by NSFC No.61303057 and Project No.2012ZX01039-004.

6. REFERENCES

[1] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire OS distributions. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 48–62, 2012.

[2] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 485–495, 2009.

[3] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[4] T. Ohta, H. Murakami, H. Igaki, Y. Higo, and S. Kusumoto. Source code reuse evaluation by using real/potential copy and paste. In *9th IEEE International Workshop on Software Clones, IWSC 2015, Montreal, QC, Canada, March 6, 2015*, pages 33–39, 2015.

[5] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in open source router software. *Computer Communication Review*, 40(3):34–40, 2010.