Preventing Backdoors In Server Applications With A Separated Software Architecture (*Extended Abstract*)

Felix Schuster, Stefan Rüster, and Thorsten Holz

Horst Görtz Institute for IT-Security (HGI), Ruhr-Universität Bochum

Abstract. We often rely on system components implemented by potentially untrusted parties. This implies the risk of *backdoors*, i.e., hidden mechanisms that elevate the privileges of an unauthenticated adversary or execute other malicious actions on certain triggers. Hardware backdoors have received some attention lately and we address in this paper the risk of software backdoors. We present a design approach for server applications that can – under certain assumptions – protect against software backdoors aiming at privilege escalation. We have implemented a proof-of-concept FTP server to demonstrate the practical feasibility of our approach.

1 Introduction

In today's computing environment, we often rely on system components that are not always implemented in-house, but by a third party. Inherently, we cannot build a trust relationship with an unknown piece of software or hardware [7] or even assume the honesty of all internal developers and designers. In reaction to this potential threat, some researchers started to investigate the feasibility to detect or mitigate backdoors in hardware components (e.g., [5,6,8]). Despite a wave of recent public discoveries of software backdoors (e.g., [1]), this kind of backdoors have received only little attention up to now. In this paper, we present a design approach to reduce the attack surface of such backdoors.

Scope. This paper examines backdoors in classic server applications that require a client to authenticate at a certain point during a session. As an example for such an application, think of a server for the *File Transfer Protocol* (FTP). From a high-level point of view, the session of a legitimate client transitions unidirectional between three states: First, the client connects to a server and at one point provides its credentials. Second, the server checks the credentials according to a specific authentication scheme. Third, the privilege of the client is escalated. Accordingly, we dissect a server application into the components *pre-auth*, *auth* and *post-auth*. In the following, we consider attackers that are able to plant various backdoors in each component, but do otherwise not possess any special capabilities such as eavesdropping on arbitrary network connections.

Types of Software Backdoors. There is a large amount of different types of backdoors that can be implemented in software [2,7]. Even classical software vulnerability classes such as *buffer overflows* can be counted as one of those types, besides more intuitive ones like *hardcoded credentials* and *hidden accounts*. From a high-level point of view, mainly two types of backdoors exist:

(A) Backdoors that are crafted to elevate the privileges of an attacker.
(B) Backdoors that trigger in the scope of legitimate sessions in order to achieve (B1) leakage of data, (B2) malfunction, or (B3) denial of service.

To illustrate this differentiation, Figure 1 schematically shows the program flow between the logical components *pre-auth*, *auth* and *post-auth* of a server application containing various exemplary backdoors of type A. In order to elevate her privileges, an attacker ultimately always needs to bypass the authentication enforcement mechanisms of an application. Thus, backdoors aiming at privilege escalation either manipulate authentication checks or circumvent them in a whole. Imagine for example the scenario of an FTP server: A knowledgeable attacker being already in possession of valid credentials to an account with privilege level x (see Figure 1) logs-in to the server. A backdoor in the *post-auth* component only triggering at a certain time-of-day automatically elevates her privileges from x to y. In contrast, backdoors of type B do not circumvent or spoil authentication,

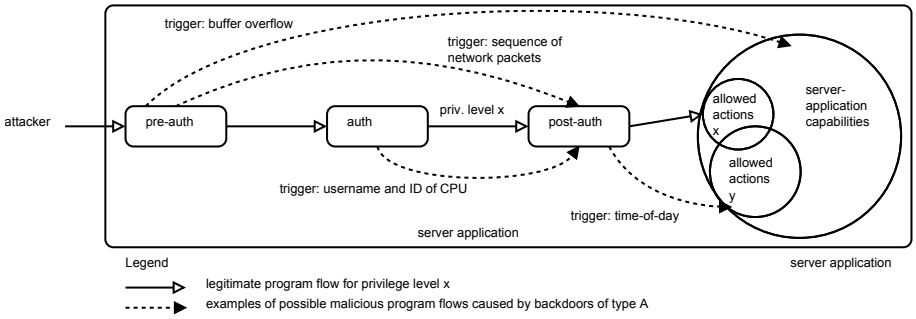

Fig. 1. Example scheme of various program flows of a server application containing backdoors of type A.

but solely operate in the context of legitimate sessions. A classic example of a backdoor of type B (as shown in Figure 2) is a compromised *auth* component that leaks the password of an unaware user on a certain trigger.

In certain cases, an attacker might not want a backdoor to be active permanently, but only in the event of certain triggers. This maximizes the chances for a backdoor to remain unnoticed during testing and actual operation of its host application. Triggers for application backdoors can either be one of the following or a combination thereof [2,8]:

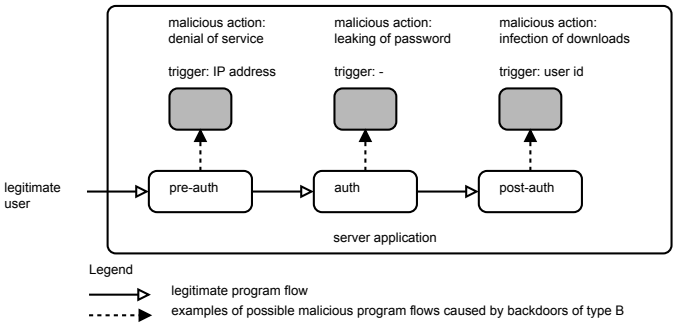

Fig. 2. Example scheme of various program flows of a server application containing backdoors of type B.

(1) *Externally supplied data* like usernames or a combination of specific values in a protocol's header
(2) *Externally induced events* like a sequence of login attempts or a time offset between network packets or other side-channels
(3) *Global or local environment* like the time of day or the ID of the local CPU

The arguably most common case is a combination of type A backdoors and type 1 triggers (see for example recent CVEs 2012-1803, 2012-4964, 2012-0209). Thus, this paper aims primarily at designing a server application that is not vulnerable to any backdoors of type A regardless of the employed triggers. We argue that this is necessarily achieved in case the following intuitive requirements are matched for a server application:

(I) The elementary transition $\boxed{\text{pre-auth}} \rightarrow \boxed{\text{auth}} \rightarrow \boxed{\text{post-auth}}$ of a session cannot be evaded.
(II) The privilege level of a session is properly enforced at any time by a secure reference monitor. There is no way of circumventing access control.
(III) The *auth* step is immune against backdoors. It is only possible to advance to *post-auth* when correct credentials are available.

Under these preconditions, imagine the attacker in Figure 1 being able to trigger code execution backdoors (which are a superset of all other possible backdoors) in all three components: Even in that case it is not possible for her to elevate her privileges from x to y or even higher.

High-level Idea. In this paper, we introduce a design approach for server applications that fulfils all of the above requirements (under certain preconditions). The risk of backdoors is thus limited to those of type B. This is mainly achieved by separating relevant parts of an application and employing a trusted reference monitor in combination with the backdoor-proof authentication system proposed by Dai et al. [2]. As a proof-of-concept, we have implemented an FTP server according to our design, and discuss the reduced attack surface. By means of this

specific implementation, we furthermore show how – depending on the actual use-case – it is possible with our approach to decrease the risk of backdoors of type B.

2 General Approach

Our approach is based on the intuition that if the requirements (I), (II) and (III) formulated in the previous section are matched, no usable backdoor elevating privileges can potentially exist in a server application. In order to achieve property (III), we need a reliable and backdoor-proof authentication system as foundation. Real-world authentication systems often go well beyond a simple password comparison and should rely on strong cryptography. Hence, it is probably naive to assume that the absence of backdoors could be entirely assured by automated or manual analysis.

A Backdoor Free Authentication. Dai et al. showed how existing *response-computable authentication* (RCA) systems can be retrofitted to become immune against backdoors and triggers of all kinds [2]. Our approach builds upon the work by Dai et al. and we outline it in the following:

The foundation of their approach is the decomposition of a conventional RCA module into two distinct components: An untrusted and probably large component that outputs an *expected response* given a password and a challenge, and a trusted and small component that compares a *received response* against the *expected response*. In case received and expected response match, the corresponding authentication is regarded as successful. Dai et al. suggest that the *response-comparison* module is manually reviewed for vulnerabilities since it should not contain much code beside a simple *memcmp()* in most cases. In contrast, the untrusted *response-computation* component is isolated using an adapted version of *Native Client* (NaCl) [9] called *NaPu*. Beside isolation and fine grained access control, *NaPu* guarantees *pure function* properties [3]. Pure functions are deterministic and side effect free. While the latter is already provided by the original *NaCl*, the former is not. To achieve determinism (i.e., here the absence of backdoors), *NaPu* renders triggers of type 3 useless by prohibiting access to the global and local environment of a server application. This is achieved through various measures, i.e., by making the x86 instruction *CPUID* unavailable and by not offering access to certain syscalls. Furthermore, triggers of type 2 (*externally induced events*) are avoided by resetting the respective program logic before each invocation. The absence of triggers of type 1 (*externally supplied data*) is ensured by automated testing before deployment: Note that an attacker can neither choose the password nor the challenge used in the calculation of an expected response. Thus, Dai et al. claim that a backdoor either triggers during testing or will only trigger in such rare cases that it is not of any practical use to an attacker.

Design of a Backdoor Free Server Application. Given this previous work by Dai et al., we show that it is possible to design a generic architecture to prevent software backdoors in server applications as depicted in Figure 3. As we will show, this architecture fulfils under certain preconditions the requirements (I), (II) and (III) necessary for the evasion of backdoors of type A. Similar to

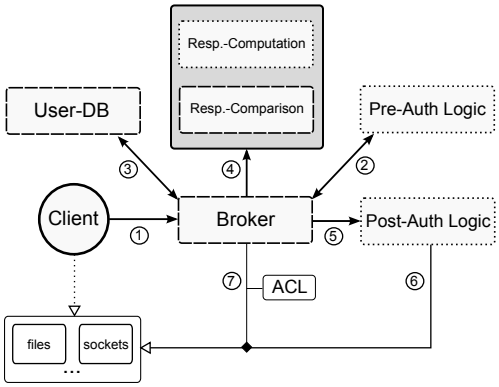

Fig. 3. High-level layout of the proposed architecture with trusted components dashed and untrusted and isolated components dotted. The gray box contains the components of the RCA as proposed by Dai et al.

how Dai et al. decomposed a RCA into distinct components, we decompose an entire server application: In Figure 3, components that are to be trusted and are backdoor free by definition are displayed as dashed boxes, while untrusted and isolated components are displayed as dotted boxes. The two components *response-computation* and *response-comparison* constituting the RCA according to Dai et al. are grouped in the gray box.

At the center of our architecture lies a trusted component called *broker* that is not application-specific and should only implement a minimum set of necessary interfaces. The purpose of the broker is to enforce authentication for every session throughout the runtime of a server application. The broker can be thought of as a classic reference monitor [4] but on application level. The (potentially backdoor containing) code of the server application resides in the two components *pre-auth* and *post-auth*. The broker starts and controls these components. It filters their requests for file, network, or similar accesses ⑥ using access control lists (ACL) ⑦. The broker initially accepts any new connections from clients ① and immediately starts transparently forwarding any communication to a newly launched and isolated *pre-auth* component ②. The sole task of this component is to act as a middleman for the client and authenticate with the broker through a secure RCA as described above[1]. The broker in turn only generates the required

[1] Note that the ACL for the *pre-auth* component (unauthenticated privilege level) should in most cases disallow access to any system resources. Though, it could for

challenge and queries a trusted database (in the simplest scenario a text file) for the password of the client ③ and invokes the authentication component with the corresponding parameters ④. The authentication process is depicted in detail in Figure 4. The reasoning behind this design is to not involve the broker in *any*

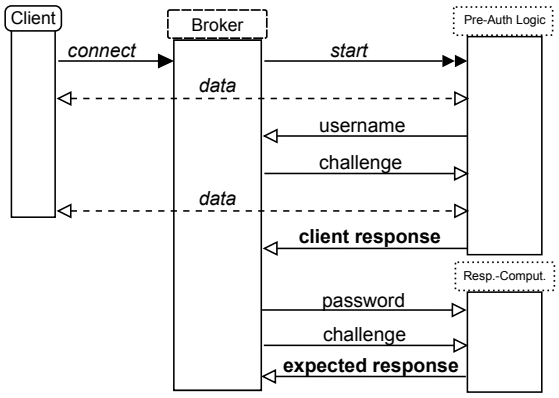

Fig. 4. Dataflow between the entities involved in the authentication process with trusted components dashed and untrusted and isolated components dotted. The trusted components *user-db* and *response-comparison* are not explicitly shown. Here they can be thought of as being a part of the trusted *broker*.

backdoor-prone protocol parsing besides the forwarding of a handful of parameters. Additionally, this construct is entirely transparent to the client, allowing the usage of legacy client software. Once a client has successfully authenticated, the broker loads the ACL corresponding to the respective authentication level and launches a new instance of the *post-auth* component of the server application ⑤ that serves the actual requests of the client.

Discussion. We claim that a server application designed according to the high-level architecture presented above meets the requirements (I), (II) and (III) under the following preconditions:

- The employed sandboxing techniques are strong.
- The trusted components *broker* and *user-db* are free of backdoors and work as expected.
- The employed RCA according to Dai et al. is secure and indeed free of usable backdoors.

It is easy to see that under the assumption of the availability of a secure broker and strong sandboxing techniques the two untrusted components *pre-auth* and

some protocols be necessary to allow for example the creation of a separate TCP/IP connection.

post-auth cannot conduct any actions despite those explicitly allowed by the ACL corresponding to a session's current authentication level. Hence, under the given preconditions requirement (II) ("the privilege level is properly enforced at any time") is matched. The same accounts for requirement (I): In case broker and sandboxes work as expected, it is not possible for a backdoor to circumvent the step of authentication.

Showing that requirement (III) ("the authentication process is immune against backdoors") holds as well is only little more complex: Though we assume the availability of an in itself secure RCA according to Dai et al., it is still possible that a compromised *pre-auth* component attempts to spoil the authentication process. Naturally the *post-auth* component cannot interfere with the authentication process since it is launched just after the authentication process was terminated. During the authentication process, the *pre-auth* component is obviously in the position to arbitrarily alter the values *username*, *challenge* and *challenge-response*. But nevertheless, there exists no way it could possibly derive privilege escalation from this circumstance, because in order to log in a client under a certain username, it always needs to pass the expected response along to the broker. Since the employed RCA is considered to be immune against backdoors [2], knowledge of the respective password is inevitably necessary in order to compute a valid response. Thus an attacker aiming at privilege escalation cannot profit from any backdoors in the *pre-auth* component more than from simply guessing passwords. Accordingly, requirement (III) is matched as well. In consequence a server application designed according to the described architecture is immune against any backdoors aiming at privilege escalation (Type A) under the aforementioned preconditions.

Further Reduction of the Attack Surface. We showed that it is not possible for an attacker to profit from any backdoor of type A in either the *pre-auth* or the *post-auth* component when a server application follows the design principles described above. What remains is the risk of backdoors performing malicious actions in the scope of legitimate sessions (Type B).

We first examine the remaining possibilities for the existence of such backdoors in the *pre-auth* component: In case the RCA protocol of a server application does not require write access to files or sockets beside the socket connection to the respective client provided by the broker, it is naturally not possible for a *pre-auth* component to leak data to a third party. Thus backdoors of type B1 can generally not exist in a *pre-auth* component in that case. It is not possible for a compromised *pre-auth* component to share data across session boundaries, as a new and isolated instance is launched for every new connection.

We claim that the only meaningful backdoor of type B2 that could possibly be installed in the *pre-auth* component performs the following malicious action: A legitimate client is secretly logged in under an account controlled by an attacker through possibly collaborating backdoors in the *pre-auth* and the *response-computation* components. For example an attacker could profit from

such a backdoor in a scenario where a higher privileged user logs in and stores confidential data. Here the attacker would get immediate access to this data.

There are various ways such a backdoor could be implemented in practice. All these ways have in common that the *pre-auth* component does not return the correct username to the broker on certain triggers, but a predefined one under which the attacker managed to create a legitimate, but probably less privileged account. The password of such an account needs in any case to be either hard-coded in the *pre-auth* or the *response-computation* component.

We claim that all variants of the attack can reliably be detected at runtime by employing the following extension to the already described basic authentication process that is depicted in Figure 4: Before sending the challenge, the broker takes a snapshot of the state of the *pre-auth* instance and its entire context and starts to write a transcript of all the messages received from the client. Besides that, the broker proceeds as normal. Once the authentication process terminates successfully, the broker does not immediately log in the respective client. Instead, the broker resets the *pre-auth* instance and replays the authentication process starting from its own challenge message. This is done in exactly the same way n times, but always with a newly generated challenge. The *pre-auth* component is not able to distinguish between the original run and the replayed ones, as long as it cannot validate the client's response on its own (which is not possible given a secure RCA) or learn the original challenge from one of the client's messages in the transcript. Thus, any practically usable backdoor designed to conduct the described attack would with respect to the size of n very likely be triggered in either none or multiple runs. In case one of the n replays terminates in a successful authentication as well, it is under the assumption that the RCA itself is strong proven, that an attempt was made to log in a client under a wrong username. Similar to the *backdoor usability testing* described by Dai et al. [2], here n needs to be chosen large enough to assure the absence of practically usable backdoors. In the following, we refer to this addition as *dynamic testing*.

What remains is the danger of type B3 backdoors (*denial of service*) in the *pre-auth* component. To the best of our knowledge, no generic mitigation is possible for such attacks. Analogously, it is to our understanding not possible to deal with a backdoors of type B in the *post-auth* component in a generic manner, since the functioning of that component is highly application specific. Instead, we assess the risk of backdoors of type B in the *post-auth* component for the FTP protocol specifically in the next section.

3 Technical Aspects and Case Study

In order to demonstrate the feasibility of our approach, we implemented a daemon for a reduced subset of FTP that complies with the architecture described above with little changes. As sandbox solution for the *pre-auth* and *post-auth* components, we chose NaCl version r9745 which we slightly extended towards our needs to have full control over file accesses and enforce mandatory access controls (MAC). The generic broker, which we tried to keep as small and

as simple as possible, is written in *C++* and consists of twelve classes composed of less than 1,300 lines of code. We decided to use a placeholder for the RCA that from the outside acts like described by Dai et al.

Discussion. Due to the nature of FTP, our *pre-auth* component does not require access to external resources such as files or additional sockets. Hence, in this case the ACL of the unauthenticated privilege level does not allow any such access, effectively preventing backdoors of type B1 (e.g., password leaks) during the authentication process. We implemented the authentication process as shown in Figure 4. Though, our broker does not yet employ the *dynamic testing* of the *pre-auth* and *response-computation* components. Our daemon is thus currently not protected against the specific backdoor of type B2 as described in the previous section, but this is planned as future work.

Our *post-auth* component is to some extent protected against backdoors of type B1 since we employ MAC on network as well as on file system accesses. Nevertheless, exploitable communication channels might exist: In order to support the *active transfer mode* of FTP, each *post-auth* ACL at least permits to establish new and direct TCP connections to the client's host on arbitrary ports. An attacker with the ability to control certain ports on a legitimate user's external host could profit from that (e.g., in a NAT scenario). Besides, information may flow between users sharing access to files. Here our ACLs' support for a no-write-down flag might limit the risk, while cutting functionality.

Unfortunately, these remaining uncontrolled communication channels can as well be exploited by backdoors of type B2 in the *post-auth* component. For example, a backdoor opening up an existent and authenticated session to a remote attacker on certain triggers is well feasible. Naturally, we cannot cope with backdoors of type B3.

4 Limitations

Even if precisely followed, the proposed architecture cannot prevent all types of backdoors in all kinds of server applications. More precisely, the architecture remains in many cases vulnerable to backdoors of type B performing malicious actions *after* the legitimate authentication of a client. Further, the proposed architecture can only be reasonably applied to server applications with independent client sessions and no continuous internal states. While server applications for well-known protocols like FTP, SMTP, or HTTP fall in this group, other protocols like IRC do not. In the case of IRC, a server necessarily needs to maintain (among many other things) a central list of all logged-in users and needs to dispatch messages among them. Such functionality can probably not be implemented using our architecture without sacrificing important security features.

5 Conclusion and Future Work

In this extended abstract, we have presented a generic architectural design for server applications that – under certain assumptions – is secure against back-

doors crafted to elevate privileges. Furthermore, our design guidelines also offer potential protection against other types of backdoors depending on the actual use case. With our approach, the attack surface for the instalment of backdoors can be significantly reduced, since only a reusable and relatively small trusted code base is required. To demonstrate the applicability of the presented architecture, we have implemented a simple FTP server accordingly. For this implementation, our approach offers protection against many types of backdoors crafted to leak or manipulate data as well.

In the future, we plan to investigate ways to extend the presented architecture in terms of immunizing more server applications against backdoors leaking and manipulating data in the scope of legitimate sessions in a more generic way. This can only be achieved reliably by either silencing or identifying corresponding backdoor triggers. Since we cannot silence all possible triggers without compromising on applicability and functionality, we need to identify them. We believe that the *dynamic testing* approach described in Section 2 can possibly be adapted and applied to the post-authentication logic of a server application to achieve this. Furthermore, we plan to retrofit an existing and full-featured FTP daemon to our architecture.

6 Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16BP12302; EUREKA-Project SASER.

References

1. RuggedCom - Backdoor Accounts in my SCADA network? You don't say..., 2012. `http://seclists.org/fulldisclosure/2012/Apr/277`.
2. S. Dai, T. Wei, C. Zhang, T. Wang, Y. Ding, Z. Liang, and W. Zou. A framework to eliminate backdoors from response-computable authentication. In *IEEE Symposium on Security and Privacy*, 2012.
3. M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in java. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.
4. C. E. Irvine. The reference monitor concept as a unifying principle in computer security education. In *In Proceedings of the IFIP TC11 WG 11.8 First World Conference on Information Security Education*, pages 27–37, 1999.
5. S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.
6. C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: Building Stealthy and Malicious Hardware. In *IEEE Symposium on Security and Privacy*, 2011.
7. K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8), Aug. 1984.
8. A. Waksman and S. Sethumadhavan. Silencing hardware backdoors. In *IEEE Symposium on Security and Privacy*, 2011.
9. B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.