GreenDog's blog

About me

Aleksei Tiurin

Просмотреть профиль

Projects

- TLS Redirection
- Reverse Proxies Chet Sheet
- Java Deserialization Cheat Sheet
- TacoTaco
- Flash AV detector
- SNI Bruter

Ярлыки

- ahk (1)
- antivirus (1)
- autobinding (1)
- browser cache (1)
- bypass (1)
- cache deception (1)
- cache poisoning (3)
- cisco (1)
- code review (1)
- cpdos (2)
- debugger (1)
- deserialization (2)
- ear (1)
- flash (1)
- group policy (1)
- http2 (1)
- j2ee (1)
- java (5)
- javadeser (1)
- javascript (1)
- mass assignment (1)
- misrouting (1)
- mitm (3)
- nbns (1)
- oracle (1)
- pentest (4)
- rce (1)
- reverse proxy (2)
- sop (1)
- spring mvc (1)
- srp (1)
- ssti (1)
- swf (1)
- tacacs (1)
- tacotaco (1)
- thymeleaf (1)
- tls (1)
- windows (3)

Архив блога

- ► 2021 (3)
- ► 2020 (2)
- ► 2019 (2)
- ► 2018 (3)
- ▼ 2017 (1)
  - ▼ марта (1)
    - Autobinding vulns and Spring MVC
- ► 2016 (2)
- ► 2015 (3)

пятница, 24 марта 2017 г.

Autobinding vulns and Spring MVC

Intro

If you don't want to read all this text, you can watch video from the 29thmeeting of Defcon Russia Group (in Russian)

There is a not so well-known vulnerability type - "autobinding", or "mass assignment". The idea this type is based on a feature that is implemented in many frameworks. It allows a framework to automatically bind HTTP request parameters to objects and make them accessible to a developer. However, an attacker can add additional HTTP request params and they will possibly be bounded to an object. Depending on a victim software and its logic, the attacker can achieve some interesting results.

The autobinding feature is pretty widespread for frameworks, which makes attack surface rather wide. However, usually it's hard to find them without knowing source code and impact of a vuln strongly depends on an application.

You can read about this type of vulns on OWASP
There are also some simple examples, so if you are not familiar with it, I recommend that you look at it.
https://www.owasp.org/index.php/Mass_Assignment_Cheat_Sheet

A "real" example of such vulnerability and some additional information for Spring MVC framework was published by Ryan Berg and Dinis Cruz in 2011 - https://o2platform.files.wordpress.com/2011/07/ounce_springframework_vulnerabilities.pdf

Something new

It's passed much time since that publication and Spring MVC now is not like it was before. It's much much cooler :)
When I was preparing tasks for the last ZeroNigths HackQuest, I wanted to provide one with an autobinding vuln to make people more familiar with this type of vulns. During the creation of the task, I've spotted an unknown/hidden variation of the autobinding vuln and I'd like to tell you about it.

As I wrote before, Spring MVC has changed. It's much smarter and has more features now. One of the new things is using annotations for doing "magic" things.
Because of them and some misunderstanding in minds, we can find an autobinding vuln in unexpected places.
Let's look at some of them and their official description (taken from here http://docs.spring.io/spring/docs/3.1.x/spring-framework-reference/html/mvc.html)

1) @ModelAttribute on a method argument
"An @ModelAttribute on a method argument indicates the argument should be retrieved from the model..."

2) @ModelAttribute on a method
"An @ModelAttribute on a method indicates the purpose of that method is to add one or more model attributes. @ModelAttribute methods in a controller are invoked before @RequestMapping methods"

3) @SessionAttribute for controller
"The type-level @SessionAttributes annotation declares session attributes used by a specific handler. This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session"

4) FlashAttribute
"Flash attributes provide a way for one request to store attributes intended for use in another."

If you are not familiar with them or don't know spring at all, don't panic, because it will be clearer with further examples :)

What do the examples 2-4 have in common? They are all somehow related to "passing" data between methods.
One of the ways to get data that was passed to the method is to use @ModelAttribute on a method argument (look at 1). However, it could lead to autobinding vuln. Why?
Because @ModelAttribute is pretty "smart". Let's look at a full description of it.

"An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the argument's fields should be populated from all request parameters that have matching names."

So, first, @ModelAttribute retrieves an object from the model (or somewhere else) and then it populates the object with a user request. Therefore, a coder expects trusted data (the object) from the model, but an attacker can change it by just sending a specially crafted request.
I've made 2 tasks for ZN HQ, and both of them contain variations of autobinding vulns. Let's have a look at what's going on there.

Sources of the tasks - https://github.com/GrrrDog/ZeroNights-HackQuest-2016

The First School of Bulimia "Edik"

The application consists of 3 "pages": registration, authentication, home page. The goal is to perform an expression language injection in the home page. The obstacle is that a user can set values only during registration process...
There is a class of User and it has 3 fields (name, password, weight) in the application.
The registration controller looks like that:

```
@InitBinder
protected void initBinder(WebDataBinder binder) {
    //logger.info("binding " );
    binder.setValidator(new UserValidator());
}

@RequestMapping(value = "/reg", method = RequestMethod.POST)
public String registerProcess(@Valid User user, BindingResult result, Model model) {
    logger.debug("result " + result );

    if (result.hasErrors()) {
        return "reg";
    }
    //old style check
    if(userService.findByName(user.getName())){
        model.addAttribute("error", "Username is already in use");
        return "reg";
    }

    logger.info("Successeful registration ! " + user);
    userService.addUser(user);
```

As we can see, the controller gets User object from a user request, validates it, and if the object is validated, the controller puts it in "DB".

```
public void validate(Object target, Errors e) {

    //logger.info("Checking ");
    User user = (User) target;

    if (user.getName() == null) {
        e.rejectValue("name", "null", "Fill your name");

    } else if (!user.getName().matches("[0-9A-Za-z]+")) {
        e.rejectValue("name", "onlynumbersletters", "Only letters and numbers
    }

    if (user.getPass() == null) {
        e.rejectValue("pass", "numeric", "Fill your password");
    } else if (!user.getPass().matches("[0-9A-Za-z]+")) {
        e.rejectValue("pass", "onlynumbersletters", "Only letters and numbers
    }

    if (user.getWeight() == null) {
        e.rejectValue("weight", "numeric", "Fill your weight. Weight has to b
```

The validating process is very strict. Because of whitelisting, we can use only figures or symbols, but we need to put special symbols in the user object! How? There is no way for the registration controller.

So, what can we do as attackers? Let's take a look at the authentication and home controller.
Authentication and home controller:

Authentication method does pretty simple things:
1) gets a username and password from a request;
2) gets a user object from the db using the username and password;
3) puts the user object in FlashAttribute and redirects to home method (sends redirect response to "/home");
So, there is no way to change the user object too.

```
@RequestMapping(value = "/authentication", method = RequestMethod.POST)
public String auth(@RequestParam String name, @RequestParam String pass, R
        Model model) {
    User user = userService.findByNamePassword(name, pass);
    if (user == null) {
        logger.debug("there is no user with name " + name);
        model.addAttribute("error", "The username or password is incorrect'
        return "/index";
    }
    attributes.addFlashAttribute("user", user);
    return "redirect:home";
}

@RequestMapping(value = "/home", method = RequestMethod.GET)
public String home(@ModelAttribute User user, Model model) {
    if (showSecret){
        model.addAttribute("firstSecret", firstSecret);
    }

    return "home";
}
```

What about the home method?
It just gets the user object from the flash attribute and shows it to us.
@modelAttribute is used in this case to get the user object, but it also can populate the user object with incoming request params! So, we can change values in the user object!
All we need to do is to authenticate (send a request to the authenticate method) and add an additional HTTP param during redirection.
So, our request will look like:

/home?name=${We_Can_Write_Here_wharever_we_want}

The goal is achieved.

Populating

There is an interesting and maybe not so obvious fact about autobinding. During populating data, Spring MVC makes changes on a field basis; it doesn't create a new object if something comes from an HTTP request. It means if there is an object from the model and only one param is received from an HTTP request, the value of only one field

(with the same name as the HTTP param) will be changed and other fields will stay the same.

Justice League

Another task. Actually, solution for this task doesn't involve using an autobinding vuln, but there is one.
The application consists of registration, authentication, home, and password recovering "pages". The latter is only one that is important for us.
Actually, recovering page is just one controller with several methods and it represents a way of creating "wizards" (multi-step forms).
Our goal for this task is to bypass authentication.

Overall logic is the following.
1) A user comes to a recovery page, inputs its username and send a request to submit the form
2) the resetHandler method receives the HTTP request. It gets a user object from the db using the username from the request. Then it puts the user object in the Model, and it automatically puts the object into a session (@SessionAttribute("user") for the controller). Then it redirects to next part of "wizard".

```
@Controller
@SessionAttributes("user")
public class ResetPasswordController {

    @RequestMapping(value = "/reset", method = RequestMethod.POST)
    public String resetHandler(@RequestParam String username, Model model)
        logger.info("Checking username " + username);
        User user = userService.findByName(username);

        if (user == null) {
            logger.info("there is no user with name " + username);
            model.addAttribute("error", "Username is not found");
            return "reset";
        }
        model.addAttribute("user", user);

        return "redirect:resetQuestion";
    }
```

3) The user is redirected to the resetViewQuestionHandler method. Actually, the method takes the user object from the session (yeah-yeah, using @ModelAttribute). It requires that object because the method has to get a custom user security question and show it in a view (however, that hasn't been implemented :)

```
@RequestMapping(value = "/resetQuestion", method = RequestMethod.GET)
public String resetViewQuestionHandler(@ModelAttribute User user) {
    logger.info("Welcome resetQuestion ! " + user);
    return "resetQuestion";
}

@RequestMapping(value = "/resetQuestion", method = RequestMethod.POST)
public String resetQuestionHandler(@RequestParam String answerReset, SessionStatus status,
        User user, Model model) {
    logger.info("Checking resetQuestion ! " + answerReset + " for " + user);

    if (!user.getAnswer().equals(answerReset)) {
        logger.info("Answer in db " + user.getAnswer() + " Answer " + answerReset);
        model.addAttribute("error", "Incorrect answer");
        return "resetQuestion";
    }

    status.setComplete();
    String newPassword = GeneratePassword.generatePassowrd(10);
    user.setPassword(newPassword);
    userService.updateUser(user);

    model.addAttribute("message", "Your new password is " + newPassword);
    return "success";
```

4) When the user sends an answer for the question, the resetQuestionHandler method handles it. The method gets the answer from "answerReset" param and compares with the value in answer field from the user object. If answers match, the method generates a new secure password and shows it to the user.
As you can see, there is no @ModelAttribute near User user (in the method argument). However, Spring MVC is smart and automatically gets value from the session. Actually, it uses the same logic: gets value from somewhere, populates it with a user request.

So, what we can do as attackers?
a) We can add "answer=any_value", when we send a request to resetViewQuestionHandler. Then our answer is populated with an object from a session. So, we can change a correct answer to any value and after that set the same value for the resetQuestionHandler method.
Therefore, we can start the recovery process for admin user, bypass answer checking, and get a new password for admin.
b) We can add "answer=any_value" on the last step too (resetQuestionHandler) and get the same results. Actually, we can change a whole object if we would like.

```
POST /justiceleague/resetQuestion HTTP/1.1
Host: localhost:8080
Cookie: JSESSIONID=2D334A1B3093196ED532A2CE27BFF7BB
Content-Type: application/x-www-form-urlencoded
Content-Length: 347

username=admin&password=123123123&isSupaAdministrata=true&email=test@test.com
&answer=1&answerReset=1
```

Session Puzzling

There is another interesting thing. When a method gets an object from a session and populates it with a user request, @SessionAttribute "forces" Spring to store this newly populated object in the session. Therefore, we are able to control values of the object that are stored in the session. How can we use it?
If there is another controller that uses the same name of session attribute and trusts it, then we can perform a Session Puzzling attack (Session Variable Overloading). https://www.owasp.org/index.php/Testing_for_Session_puzzling_(OTG-SESS-008) . As far as I remember, the documentation says that a session attribute (created using

@SessionAttribute) is limited to a controller, but in practice, we can use it in other controllers too.

Real Examples

I was looking for real examples of such issues on GitHub and in articles about Spring MVC and even found some. But:
1) examples identified on GitHub look like someone's experiments with Spring MVC (not like real stuff)
2) there are some articles that recommend "dangerous" using of @ModelAttribute, but their examples are too simple and there is no potential impact.

Detection

At first glance, finding autobinding vulns using the "blackbox" approach looks impossible. Nevertheless, both of tasks were solved, both autobinding vulns were found. Respect to these people :)

There are some things that can help us to find such type of vulns:
1) often, a param name is equal to the name of a field of an object (but not necessary, because it's configurable). As the fields are often named in specific way, we can distinguish them. Of note, autobinding can be used with hashmaps and arrays.
2) When autobinding in a controller's method is used and when we send two parameters with the same name, the value in the object will be a concatenation of parameters.
Example:
Request with params:

```
?name=text1&name=text2
```

Result:

```
ObjectWithNameField.name = text1,text2
```

3) As soon as we've collected all param names, we can send them to all entry points (URLs), even to those that, at first glance, don't accept params (like resetViewQuestionHandler), and check if replies are different or the same as without params.

Conclusion

I've not shown an example related to incorrect using of "@ModelAttribute on a method", but something similar could happen in this case too.
As you can see, the main idea, is based on the fact that a programmer thinks that he or she gets an object from a trusted place, but in practice, the object can be modified by an attacker.

I'm not sure, that I've correctly described causes of such a vuln, why and how Spring MVC behaves in this way.

It's hard to say exactly how common this variation of autobinding vuln is (but it definitely can be somewhere ;) . In general, autobinding vulns are pretty widespread, because of the feature they are based on. Moreover, the autobinding is not just about HTTP params, theoretically, any incoming data (e.g. JSON or XML) can be converted and then populated. However, a possibility of exploitation and impact strongly depend on many things ranging from using annotations and names of attributes to business logic of an application.

Sources of the tasks - https://github.com/GrrrDog/ZeroNights-HackQuest-2016

Автор: Aleksei Tiurin на 04:30

Ярлыки: autobinding, java, mass assignment, spring mvc

1 комментарий:

rusyasoft 26 мая 2019 г. в 23:35

Этот комментарий был удален автором.

Ответить

Чтобы оставить комментарий, нажмите кнопку ниже и войдите с

Следующее Главная страница Предыдущее

Подписаться на: Комментарии к сообщению (Atom)