Noname manuscript No.
(will be inserted by the editor)

XSnare Signature Language Description

José Carlos Pazos

1 XSnare Design

A signature is specified as a JavaScript object. Each of the fields in this object captures an aspect of the signature's behaviour, including website identification, type of request, injection endpoints, and sanitization methods. Some fields are optional, as they are either used for added signature loading precision, or only need to be defined based on the value of another field. Required fields are used to identify the type of request and the injection point details. Additionally, some fields, like the sanitization method have default options. Table 1 lists these fields and their semantics. Our extension includes several pre-packaged signatures, so a signature developer can refer to a large set of example signatures.

When dealing with dynamic requests additional details need to be specified. If the value of *type* in Table 1 is 'listener', the signature will have an additional field called *listenerData*, which acts as a nested signature with fields that identify a particular dynamic request. A developer can optionally specify the request's injection points using this nested signature. Table 2 describes this sub-signature format.

1.1 Design language considerations

While our signature language has a strong basis on the inner workings of HTML and XSS, we not only wanted to make a language strong enough to cover a wide variety of vulnerabilities, but also one that would feel intuitive for an analyst and would not overwhelm them with unnecessary details. In this sense, we've had to be rather restrictive in the constructs we allow.

For most of the signature fields, we've taken careful consideration into its design:

Unnamed institute

Signature fields and descriptions
url. Optional (string) If the exploit occurs in a specific URL or subdomain, this is defined as a string, e.g., `/wp-admin/options-general.php?page=relevanssi%2Frelevanssi.php`, otherwise null.
software. Optional (string) The software framework the page is running, e.g., 'WordPress'. Some pages might not have a framework, i.e., handcrafted pages. Note that software probes, are required for specific software. Probes are built-in application detectors in the extension. If a probe is not available, in our current implementation, the software will not be correctly identified, as there is currently no built-in way to add probes to the extension.
softwareDetails. Optional (string) If the *software* field has been given a value, this field can provide further information about when to load a signature for this software for added precision; since a vulnerability could occur in a website running the base software, or a plugin for this software (thus, this field remains optional even if *software* has been given a value). For WordPress and similar CMS's, these are plugin names as depicted in the HTML of a page running the plugin. For example, most WordPress plugins can be identified by a script element with the substring *wp-content/plugins/PluginName*. In this case, the value of *softwareDetails* would be *PluginName*. Note that this value should be as precise as possible, because it will determine when the signature is loaded. Furthermore, this depends on the value of *software*. In particular, probes for both the software and the software details need to be implemented.
version. Optional (string) The version number of the software/plugin/page, used for versioning. Note that versioning in general is hard and can only be done with precise information or elevated permissions.
type. Required (string) Parameter for describing the signature type, 'string' describes a basic signature, 'listener' describes a signature with a listener for additional network requests. See Table 2 for more details on listener specifications.
listenerData. Only required if *type* is 'listener' (string) A field for specifying the identification information for a dynamic request. This is written as a nested signature. See Table 2 for more details on listener specifications.
sanitizer. Optional (string) The sanitizer to be used, "DOMPurify", "escape", or "regex". The default is DOMPurify.
sanitizerConfig. Optional for DOMPurify and escape. Required for regex. (string for regex, tuple of strings for escaping, or JavaScript DOMPurify configuration) Additional configuration parameters to go along with the chosen sanitizer. For example, DOMPurify allows configuration through its API (i.e., DOMPurify.sanitize(dirty, sanitizerConfig). For 'escape', an additional escaping pattern can be provided, in the form of array, where the elements are the arguments of JavaScript's 'string.replace' method. For 'regex', this should be the pattern to match with the injection point content.
typeDet. Required (string) A string describing the recurrence of injection points throughout the document, e.g., 'single-unique'. This is specified in the format 'occurrence-uniqueness': 'occurrence' has values single/multiple, which describes the existence of one or multiple independent injection points; the 'uniqueness' has values unique/several, specifying whether an injection point occurs once or several times throughout the document.
endPoints. Required (array of tuples) An array of startpoint and endpoint tuples, specified as strings for regex matching.
endPointsPositions. Optional (array of tuples) An array of integer tuples. These are optional but useful when the one of the endPoints HTML are used throughout the whole page and appear a fixed number of times. For example: if an injection ending point happens on an element <h3 class='my-header'>, this element might have 10 appearances throughout the page. However, only the 4th is an injection ending point. The signature would specify the second element of the tuple to be 7, as it would be the 7th such item in a regex match array (using 1-based indexing), counting from the bottom up. For ending points, we have to count from the bottom up because the attacker can inject arbitrarily many of these elements before it, and vice versa for starting points.

Table 1: XSnare Language Description. Each row describes a field in the signature object. Fields can be required or optional, sometimes based on values of other fields. Additionally, some optional fields have default non-null values.

listenerData **sub-signature fields and descriptions**
listenerType. Required (string) The type of network listener as defined by the WebRequest API [1] (e.g., 'script', 'XHR', etc.)
listenerMethod. Required (string) The request's HTTP method, for example "GET" or "POST".
requestUrl. Required (string) The URL of the dynamic request target. Note that this is different from the page's URL. For example, an XHR could be fetched from the '/my-ajax.php' subdomain

Table 2: Additional fields for *listenerData* when a *type* of 'listener' has been specified in a signature. Injection endpoint information (*typeDet*, *endPoints*, *endPointsPositions*) and sanitization information (*sanitizer*, *sanitizerConfig*) can be specified either in this subsignature or in the parent signature. A listener could even have another listener nested into it, by setting the *type* and *listenerData* fields. We have omitted the repetition of those fields in this table.

- **url**: While our evaluation has focused mostly on WordPress sites, and in general in CMSs, this does not mean our extension only applies to those. In fact, it would be easier for us to detect vulnerabilities in some other pages, for example, by URL. This way, we can skip the probing and even the versioning. Even in the case of WordPress sites running vulnerable plugins, we found that many of them manifested themselves only in a subpage of the site. A signature developer can specify this page so as to not affect other pages which would also trigger the signature.
- **software**: Probes are an important component of our software identification functionality. While we could have offloaded software identification to the signature writer, in practice we found that much of the same software is prone to having vulnerabilities. Naturally, this is bound to occur in an open development environment, like in many CMS systems. Thus, we believe that implementing the probes directly in our extension alleviates the workload for the developer. As such, one need only specify the name of the targeted web software as a string.
- **softwareDetails**: As this field depends directly on the 'software', it follows a similar design choice: software specific probes are able to identify software details, for example, plugins in WordPress, as these are most often identified in the same way in the HTML.
- **version**: We discuss the details and complications that arise from versioning in our paper.
- **sanitizer**: We provide three different methods: DOMPurify, regex and escape. While DOMPurify is the most powerful out of the three, we found it to be unnecesarily restrictive in many cases in practice. We consider it the default option, but a developer might deem the other methods better suited for a particular scenario.

- **sanitizerConfig**: Even though we consider DOMPurify as the default sanitization method, it also provides much more fine-grained control over its sanitization [2]. The developer can take advantage of this to tune the signature. For escaping, the default behaviour is to convert all of the injection point into a harmless HTML string by putting it inside a div's textContent property [?]. The developer can specify a pattern for escaping, which will use JavaScript's *replace* method on the injection point. Regex matching does not have a default behaviour, this field specifies which pattern to match. If the pattern matches the injection point text, it will leave it as is. Otherwise, it will remove the content entirely. These different behaviours correspond to distinct requirements: DOMPurify is a more powerful, but potentially less precise sanitizer, escaping is for when an injection point should be allowed certain string content that can be sanitized by taking out special characters (e.g., a string without "<, >, [,], etc."), and regex is for when a restrictive pattern should be met (e.g., only alphanumeric characters).
- **typeDet**: We went through several iterations to determine the most concise way to specify the cases where one page might have several injections. We eventually decided that it could be reduced to four fundamental cases, dictated by two binary options. That is, the same injection can occur once or several times throughout the HTML, its 'occurrence' (e.g., in the case of a table with several elements), and there can be several distinct injections in the same page (e.g., one page suffering from several different vulnerabilities).
- **endPoints**: As our injection points are just a StartPoint and EndPoint couple, we decided that these should be specified as list of tuples (2-element arrays). Of course, this is where the developer has to do the most work to effectively identify the injection points. Even a small mistake could result in a missed vulnerability.
- **endPointsPositions**: Even similar strings might not be subject to the same vulnerabilities. For example, an <input class="some-input"> element might only be subject to an exploit if it occurs inside of another element. An analyst familiarized with HTML/CSS might find it easier to specify these locations with Selectors. However, tracking these positions is best done in a parsed HTML site, not a text string. Unfortunately, we cannot do this because it might result in element rearrangement. Therefore, we decided to specify these positions as arrays of indices. We believe that this is one of the most tedious aspects of writing a signature. Fortunately, this only occurs in a small subset of vulnerabilities. In our studied sites, we used this field in 7 signatures (out of 64 we wrote).
- **listenerDetails**: When dealing with dynamic requests, we decided to have only the minimal required information for effective request identification. All of these fields should be easy to specify for an analyst. For flexibility, we decided to allow many of the top-level signature fields as well.

[2] `https://github.com/cure53/DOMPurify/tree/main/demos#what-is-this`