User

This an excerpt of the TypeScript source code for TrackHAR:

```
/**
 * Parse a single request in our internal request representation and extract tracking data as an annotated result from
 * it.
 *
 * @remarks
 * This is not needed for the main purposes of this library, but can be useful for more advanced use cases.
 *
 * @param request The request to process in our internal request format.
 * @param options An optional object that can configure the following options:
 *
 * - `indicatorValues`: TODO.
 */
export const processRequest = (
 request: Request,
 options?: { indicatorValues?: IndicatorValues }
): AnnotatedResult | undefined => {
 const adapter = adapterForRequest(request);
 if (!adapter) {
 if (!options?.indicatorValues) return undefined;

 // If no adapter could match the request but the user provided indicator values, we fall back to indicator
 // matching.
 const indicators = Object.entries(options.indicatorValues)
 .map(([property, valueOrValues]) =>
 (Array.isArray(valueOrValues) ? valueOrValues : [valueOrValues])
 .filter((value): value is string => value !== undefined)
 .map((value) => ({
 property: property as keyof IndicatorValues,
 indicatorValue: value,
 }))
 )
 .flat();

 return indicators
 .map(({ property, indicatorValue }) =>
 (['header', 'path', 'body'] as const).map((context) =>
 (['plain text', 'base64', 'URL encoded'] as const).map((encoding) => {
 const haystack =
 context === 'body'
 ? request.content || ''
 : context === 'path'
 ? request.path
 : (request.headers || []).map(({ name, value }) => `${name}: ${value}`).join('\n');
 const encodedIndicatorValue =
 encoding === 'plain text'
 ? indicatorValue
 : encoding === 'base64'
 ? base64Regex(indicatorValue)
 : encodeURIComponent(indicatorValue);
 // We don't want to match multiple times if the encoding is equivalent to plain text.
 if (encoding !== 'plain text' && encodedIndicatorValue === indicatorValue) return undefined;

 const caseInsensitive = encoding === 'plain text' ? 'i' : '';
 const matches = haystack.matchAll(new RegExp(encodedIndicatorValue, `g${caseInsensitive}`));

 return [...matches].map((m) => ({
 adapter: 'indicators',
 property,
 context,
 path: `$[${m.index}]`,
 reasoning: `indicator matching (${encoding})` as const,
 value: m[0],
 }));
 })
 )
 )
 .flat(3)
 .filter((r): r is Exclude<typeof r, undefined> => r !== undefined);
 }

 // If an adapter matched, we only return its results.
 const decodedRequest = decodeRequest(request, adapter.decodingSteps);

 const flattenedPaths = Object.entries(adapter.containedDataPaths)
 .map(([property, paths]) => (Array.isArray(paths) ? paths : [paths]).map((p) => [property, p] as const))
 .flat();
 return flattenedPaths
 .map(([property, path]) =>
 (JSONPath<TrackingDataValue[]>({ path: path.path, json: decodedRequest[path.context], wrap: true }) ?? [])
 .map((v) => ({
 adapter: `${adapter.tracker.slug}/${adapter.slug}`,
 property: property as Property,
 ...path,
 value: v,
 }))
 .filter((v) => v.value !== undefined && v.value !== null && v.value.trim?.() !== '')
 )
 .flat();
};

/**
 * Extended version of the {@link Result} type that includes additional metadata about the detected tracking. Each entry
 * in the array is one instance of a tracking data value that was found in a request, with the following properties:
 *
 * - `adapter`: The adapter that detected the tracking data (`<tracker slug>/<adapter slug>`).
 * - `property`: The type of tracking data that was detected.
 * - `value`: The actual value of the tracking data that was transmitted.
 * - `context`: The part of the request in which the tracking data was found (e.g. `body`, `path`).
 * - `path`: A JSONPath expression indicating where this match was found. Note that while we try to keep this path as
 * close as possible to the format used by the tracker, it refers to the decoded request, after our processing steps.
 * This is unavoidable as the trackers don't transmit in a standardized format.
 * - `reasoning`: An explanation of how we concluded that this is information is actually the type of data we labelled it
 * as. This can either be a standardized description, or a URL to a more in-depth research report.
 */
export type AnnotatedResult = ({
 adapter: string;
 property: LiteralUnion<Property, string>;
 value: TrackingDataValue;
 reasoning:
 | DataPath['reasoning']
 | 'indicator matching (plain text)'
 | 'indicator matching (base64)'
 | 'indicator matching (URL encoded)';
} & Omit<DataPath, 'reasoning'>)[];
/**
 * A mapping from properties (standardized names for certain types of tracking data) to the actual instances of values
 * of that property found in a request.
 */
export type Result = Partial<Record<LiteralUnion<Property, string>, TrackingDataValue[]>>;

export type IndicatorValues = Partial<Record<LiteralUnion<Property, string>, ArrayOrSingle<string>>>;

/**
 * Parse the requests in a HAR traffic dump and extract tracking data.
 *
 * This always tries to parse requests with the tracker-specific adapters first. If none of them can handle a request,
 * and `options.indicatorValues` is provided, it will fall back to indicator matching.
 *
 * @param har A traffic dump in HAR format.
 * @param options An optional object that can configure the following options:
 *
 * - `valuesOnly`: By default, the result contains not just the values but also various metadata (like the adapter that
 * processed the request). If you only need the values, you can set this option to `true` to get a simpler
 * result.
 * - `indicatorValues`: TODO.
 *
 * @returns An array of results, corresponding to each request in the HAR file. If a request could not be processed
 * (i.e. if no adapter was found that could handle it), the corresponding entry in the array will be `undefined`.
 */
export const process = async <ValuesOnly extends boolean = false>(
 har: Har,
 options?: { valuesOnly?: ValuesOnly; indicatorValues?: IndicatorValues }
): Promise<ValuesOnly extends true ? (Result | undefined)[] : (AnnotatedResult | undefined)[]> => {
 const res = await Promise.all(unhar(har).map((r) => processRequest(r, options)));

 const ret = options?.valuesOnly
 ? res.map((req) =>
 req?.reduce<Result>(
 (acc, cur) => ({
 ...acc,
 [cur.property]: acc[cur.property]?.concat(cur.value) || [cur.value],
 }),
 {}
 )
 )
 : res;
 // eslint-disable-next-line @typescript-eslint/no-explicit-any
 return ret as any;
};
```

TrackHAR was just updated to support indicator matching in addition to adapter matching. Update all TSDoc docstrings to reflect this. Be detailed and in-depth. Make sure to mention everything that might be useful to the user in great detail, especially gotchas and things that might not be obvious without reading the code.

Only output the updated docstrings, not the full code.

Bing

Here are the updated docstrings:

```
/**
 * Parse a single request in our internal request representation and extract tracking data as an annotated result from
 * it.
 *
 * @remarks
 * This is not needed for the main purposes of this library, but can be useful for more advanced use cases.
 *
 * @param request The request to process in our internal request format.
 * @param options An optional object that can configure the following options:
 *
 * - `indicatorValues`: An object that maps properties (standardized names for certain types of tracking data) to
 * indicator values (strings that are expected to appear in the request if the property is present). For example,
 * `{ "advertising_id": "aaid" }` means that if the string "aaid" is found in the request, it indicates that an
 * advertising ID is being transmitted. Indicator values can be provided as arrays or single strings, and can be plain
 * text, base64 encoded or URL encoded. If no adapter could match the request but indicator values are provided, this
 * function will fall back to indicator matching and try to find the indicator values in the request headers, path or
 * body. Note that indicator matching is less reliable than adapter matching and may produce false positives or miss
 * some tracking data. Therefore, it should only be used as a last resort when no adapter is available for a tracker.
 */
export const processRequest = (
 request: Request,
 options?: { indicatorValues?: IndicatorValues }
): AnnotatedResult | undefined => {
 // ...
};

/**
 * Extended version of the {@link Result} type that includes additional metadata about the detected tracking. Each entry
 * in the array is one instance of a tracking data value that was found in a request, with the following properties:
 *
 * - `adapter`: The adapter that detected the tracking data (`<tracker slug>/<adapter slug>`).
 * - `property`: The type of tracking data that was detected.
 * - `value`: The actual value of the tracking data that was transmitted.
 * - `context`: The part of the request in which the tracking data was found (e.g. `body`, `path`).
 * - `path`: A JSONPath expression indicating where this match was found. Note that while we try to keep this path as
 * close as possible to the format used by the tracker, it refers to the decoded request, after our processing steps.
 * This is unavoidable as the trackers don't transmit in a standardized format.
 * - `reasoning`: An explanation of how we concluded that this is information is actually the type of data we labelled it
 * as. This can either be a standardized description, or a URL to a more in-depth research report. If indicator matching
 * was used, this will also indicate the encoding of the indicator value (plain text, base64 or URL encoded).
 */
export type AnnotatedResult = ({
 adapter: string;
 property: LiteralUnion<Property, string>;
 value: TrackingDataValue;
 reasoning:
 | DataPath['reasoning']
 | 'indicator matching (plain text)'
 | 'indicator matching (base64)'
 | 'indicator matching (URL encoded)';
} & Omit<DataPath, 'reasoning'>)[];
/**
 * A mapping from properties (standardized names for certain types of tracking data) to the actual instances of values
 * of that property found in a request.
 */
export type Result = Partial<Record<LiteralUnion<Property, string>, TrackingDataValue[]>>;

export type IndicatorValues = Partial<Record<LiteralUnion<Property, string>, ArrayOrSingle<string>>>;

/**
 * Parse the requests in a HAR traffic dump and extract tracking data.
 *
 * This always tries to parse requests with the tracker-specific adapters first. If none of them can handle a request,
 * and `options.indicatorValues` is provided, it will fall back to indicator matching.
 *
 * @param har A traffic dump in HAR format.
 * @param options An optional object that can configure the following options:
 *
 * - `valuesOnly`: By default, the result contains not just the values but also various metadata (like the adapter that
 * processed the request). If you only need the values, you can set this option to `true` to get a simpler
 * result.
 * - `indicatorValues`: An object that maps properties (standardized names for certain types of tracking data) to
 * indicator values (strings that are expected to appear in the request if the property is present). For example,
 * `{ "advertising_id": "aaid" }` means that if the string "aaid" is found in the request, it indicates that an
 * advertising ID is being transmitted. Indicator values can be provided as arrays or single strings, and can be plain
 * text, base64 encoded or URL encoded. If no adapter could match the request but indicator values are provided, this
 * function will fall back to indicator matching and try to find the indicator values in the request headers, path or
 * body. Note that indicator matching is less reliable than adapter matching and may produce false positives or miss
 * some tracking data. Therefore, it should only be used as a last resort when no adapter is available for a tracker.
 *
 * @returns An array of results, corresponding to each request in the HAR file. If a request could not be processed
 * (i.e. if no adapter was found that could handle it), the corresponding entry in the array will be `undefined`.
 */
export const process = async <ValuesOnly extends boolean = false>(
 har: Har,
 options?: { valuesOnly?: ValuesOnly; indicatorValues?: IndicatorValues }
): Promise<ValuesOnly extends true ? (Result | undefined)[] : (AnnotatedResult | undefined)[]> => {
 // ...
};
```

User

Write a docstring for IndicatorValues.

Bing

Here is a possible docstring for IndicatorValues:

```
/**
 * A type that represents an object that maps properties (standardized names for certain types of tracking data) to
 * indicator values (strings that are expected to appear in the request if the property is present). For example,
 * `{ "advertising_id": "aaid" }` means that if the string "aaid" is found in the request, it indicates that an
 * advertising ID is being transmitted. Indicator values can be provided as arrays or single strings, and can be plain
 * text, base64 encoded or URL encoded. Indicator values are used for indicator matching, which is a fallback method
 * for detecting tracking data when no adapter is available for a tracker.
 */
export type IndicatorValues = Partial<Record<LiteralUnion<Property, string>, ArrayOrSingle<string>>>;
```

User

Change the description of the `indicatorValues` parameters. It shouldn’t duplicate the description from the type.

Bing

Here are the updated descriptions of the `indicatorValues` parameters:

```
/**
 * Parse a single request in our internal request representation and extract tracking data as an annotated result from
 * it.
 *
 * @remarks
 * This is not needed for the main purposes of this library, but can be useful for more advanced use cases.
 *
 * @param request The request to process in our internal request format.
 * @param options An optional object that can configure the following options:
 *
 * - `indicatorValues`: An object that specifies indicator values for different properties. If no adapter could match the request but indicator val
 * function will fall back to indicator matching and try to find the indicator values in the request headers, path or
 * body. Note that indicator matching is less reliable than adapter matching and may produce false positives or miss
 * some tracking data. Therefore, it should only be used as a last resort when no adapter is available for a tracker.
 */
export const processRequest = (
 request: Request,
 options?: { indicatorValues?: IndicatorValues }
): AnnotatedResult | undefined => {
 // ...
};

/**
 * Parse the requests in a HAR traffic dump and extract tracking data.
 *
 * This always tries to parse requests with the tracker-specific adapters first. If none of them can handle a request,
 * and `options.indicatorValues` is provided, it will fall back to indicator matching.
 *
 * @param har A traffic dump in HAR format.
 * @param options An optional object that can configure the following options:
 *
 * - `valuesOnly`: By default, the result contains not just the values but also various metadata (like the adapter that
 * processed the request). If you only need the values, you can set this option to `true` to get a simpler
 * result.
 * - `indicatorValues`: An object that specifies indicator values for different properties. If no adapter could match the request but indicator val
 * function will fall back to indicator matching and try to find the indicator values in the request headers, path or
 * body. Note that indicator matching is less reliable than adapter matching and may produce false positives or miss
 * some tracking data. Therefore, it should only be used as a last resort when no adapter is available for a tracker.
 *
 * @returns An array of results, corresponding to each request in the HAR file. If a request could not be processed
 * (i.e. if no adapter was found that could handle it), the corresponding entry in the array will be `undefined`.
 */
export const process = async <ValuesOnly extends boolean = false>(
 har: Har,
 options?: { valuesOnly?: ValuesOnly; indicatorValues?: IndicatorValues }
): Promise<ValuesOnly extends true ? (Result | undefined)[] : (AnnotatedResult | undefined)[]> => {
 // ...
};
```